USENIX

VERY HIGH LEVEL LANGUAGES (VHLL) PROCEEDINGS

Autumn 1994

U
US
USE
U NIX
USENIX

Very High Level Languages

Symposium Proceedings

October 26-28, 1994
Santa Fe, New Mexico

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $23 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$10 per copy for postage (via air printed matter).

ISBN 1-880446-65-0

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

Proceedings of the USENIX Symposium

on

Very High Level Languages (VHLL)

October 26-28, 1994
Santa Fe, New Mexico, USA

Table of Contents

Very High Level Languages Symposium

October 26-28, 1994
Santa Fe, New Mexico

Program Committee

An anecdote about ML type inference

Andrew Koenig (`ark@research.att.com`)

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

ML strongly supports functional programming; its programmers tend to avoid operations with side effects. Thus, for example, instead of mutable arrays, ML programmers usually use lists. These lists behave much like those in Lisp, with the added requirement that all elements have the same type.[1] ML lists are not themselves mutable, although the individual elements can be of a (uniform) mutable type. In Lisp terms, ML lists can optionally support `RPLACA` but not `REPLACD`. Although it is possible to define a type in ML that would support `REPLACD` as well, ML programmers usually don't do this unless they need it.

The Lisp functions `CAR` and `CDR` have ML equivalents called `hd` and `tl` and ML represents the Lisp `CONS` function by a right-associative infix operator `::` (pronounced ''cons'').

While learning ML, I thought it would be a nice exercise to try to write a sort function. Although Standard ML does not define arrays, the implementation I was using (Standard ML of New Jersey) did have them. Nevertheless, I felt it would be a more useful exercise to try to write a purely functional sort program: one that operated on lists, rather than arrays, and had no side effects at all.

Strategy

How does such a function behave? It must evidently take a list as argument and return a sorted list as its result. Moreover, since we don't know the type of the list elements, we had better give the function a second argument, namely a function to use for comparisons.

The desired function, then, should have a type[2] of

```
('a * 'a -> bool) -> 'a list -> 'a list
```

where `'a` represents any type, as usual, so that

```
sort (op <) [3,1,4,1,5,9]
```

would yield

```
[1,1,3,4,5,9]
```

Such a function will be structured something like this:

```
fun sort less x =
     let (* local function definitions go here *)
      in (* some expression *)
     end
```

so as to make the comparison function `less` and the argument `x` available to the local functions without further formality. To make these local functions easier to write, we can therefore start by temporarily defining an appropriate comparison function for integers:

1. This restriction is typically not much of an impediment, because any set of specific types can be bundled into a single type called a *datatype*. ML datatypes are somewhat like a semantically safe version of C unions.

2. Experienced ML programmers will note here that `sort` is actually a curried function, so as to allow partial application. For instance, it is nice to be able to treat `sort (op < : int*int->bool)` as a single function without having to supply a list to sort at the same time.

```
fun less(x: int, y) = x < y
```

or, more directly:

```
val less: int * int -> bool = op <
```

In either case, we can now use `less` as an integer comparison function while writing the inner functions that will make up our sort; when we're ready to combine them, the name `less` will then smoothly refer to the one that was passed as an argument to `sort` rather than the one we just defined.

How does one sort a list? The strategy that presented itself was a recursive merge sort: split the list into two pieces, sort the halves, then merge the sorted halves. That suggests that `sort` should have two local functions called `split` and `merge`. With that in mind, I decided to try writing the inner functions.

Tactics

I did not immediately see a good way to split a list, so I began by writing `merge`. The `merge` function takes two arguments, each of which is a sorted list. If either of the arguments is empty, the result is the other. Otherwise, we compare the initial elements of the two lists; the result is the smaller of the two followed by the merge of what's left.

That's easier to express as a program than in prose:

```
fun merge(nil, x) = x
  | merge(x, nil) = x
  | merge(x as h::t, y as h'::t') =
       if less(h, h') then h::merge(t,y) else h'::merge(t',x)
```

A few observations may make it easier to follow this program. It is written as a series of alternatives, or *clausal definitions*. These alternatives are always tested in sequence. In other words, `merge` first checks if its first argument is `nil`; if so, it returns the second. Next it checks if the second argument is `nil`; and returns the first argument in that case.

The third clause is more interesting. Note first that an apostrophe in an identifier is just a letter, so that `h'` is pronounced ''h prime.'' Next note the *layered patterns* that appear in the formal parameters. Saying `x as h::t` means ''the formal parameter is `x`, which incidentally must be of the form `h::t`.'' Recall that `::` is like CONS in lisp, so `h` is the first element of the list `h::t` and `t` is the rest of it. The effect of `x as h::t`, then, is to allow `x` to refer to the entire argument and simultaneously to allow `h` and `t` to refer to the first element and the rest of the list, respectively. This is guaranteed to be possible because the earlier clauses eliminated the case where either argument was `nil`.

The `merge` function as shown above seems to work, and indeed it did not take much experimentation to convince me that I had it right. Next I had to tackle `split`.

The type of `split` is straightforward: it takes a list as input and returns two lists:

```
'a list -> 'a list * 'a list
```

Now comes the hard part: how does one go about writing it?

The obvious way to split a list is to take the first half and put it one place and the second half and put it somewhere else. The problem with that is that we don't know how big the list is without traversing it; it is then necessary to traverse it *again* to do the actual splitting.

It would be nice to avoid this two-pass property. After thinking about it for a while, I realized that the problem was like to dealing a deck of cards into two equal piles. If I want to do that by cutting the deck, I must count the cards first, but instead I can just deal cards alternately onto the piles until I run out.

That insight makes the solution much easier. Splitting an empty list gives a pair of empty lists. Splitting a list with only one element gives a one-element list and an empty list. Splitting a list with more than one element lets us take the first two elements and put them on the beginning of the result lists, using `split` recursively to generate the rest of the result. Again it's easier to write than to describe:

```
fun split(nil) = (nil, nil)
  | split([x]) = ([x], nil)
  | split(x::y::tail) =
      let val (p, q) = split(tail)
       in (x::p, y::q)
      end
```

Again, this function works as written and a little experimentation convinced me that I had it right.[3]

Putting it all together

With `merge` and `split` safely in place, it seemed a simple matter to write `sort` itself. We merely follow the earlier description:

```
fun sort(nil) = nil
  | sort(x) =
      let val (p, q) = split(x)
       in merge(sort(p), sort(q))
      end
```

and indeed, when I typed in this program, the compiler accepted it.

I noticed something very curious, however: I expected the type of `sort` to be

```
int list -> int list
```

because the previous definition of `less` constrains this particular `sort` to work only on lists of integers. Much to my surprise, the compiler reported a type of

```
'a list -> int list
```

In other words, this `sort` function accepts a list of any type at all and returns a list of integers.

That is impossible. The output must be a permutation of the input; how can it possibly have a different type? The reader will surely find my first impulse familiar: I wondered if I had uncovered a bug in the compiler!

After thinking about it some more, I realized that there was another way in which the function could ignore its argument: perhaps it sometimes didn't return at all. Indeed, when I tried it, that is exactly what happened: `sort(nil)` did return `nil`, but sorting any non-empty list would go into an infinite recursion loop.

Once I saw this, it was not hard to figure out why. Suppose, for example, that we are sorting a one-element list. We split it into a one-element list and an empty list and then try recursively to sort the one-element list. That recursion makes no progress, so the computation never terminates.

The fix is simple: add a clause to `sort` to handle a single-element argument. If the argument is more than one element, `split` will certainly reduce it, so that should be sufficient:

3. ML experts will note that the `split` function can more efficiently be written this way:

```
fun split x =
        let fun loop(p, q, nil) = (p, q)
              | loop(p, q, [a]) = (a::p, q)
              | loop(p, q, a::b::rest) = loop(a::p, b::q, rest)
         in loop(nil, nil, x)
        end
```

This function is faster because it is a tail recursion. It also has the side effect of reversing the list on the way, thus yielding an unstable sort. It is an interesting exercise to restore the stability of the sort by reversing the list *again* during merging. This requires the added finesse that the comparison function must have its sense reversed during odd-numbered recursions, because it is then being used to sort a reversed list. The result can be a substantially faster sort, but is certainly harder to understand.

```
    fun sort(nil) = nil
      | sort([a]) = [a]
      | sort(x) =
          let val (p, q) = split(x)
           in merge (sort(p), sort(q))
          end
```

The compiler reports the type of this function as

```
    int list -> int list
```

as expected, and again a few experiments show convincingly that it does what it should.

It is now easy to write the general sort function as promised earlier. We need merely to make less an argument to sort, account for that in the recursive calls, and make the merge and split functions local:

```
    fun sort less nil = nil
      | sort less [a] = [a]
      | sort less x =
        let fun merge (nil, x) = x
                | merge (x, nil) = x
                | merge (x as h::t, y as h'::t') =
                      if less(h, h') then h::merge(t,y) else h'::merge(t',x)
            fun split(nil) = (nil, nil)
                | split([x]) = ([x], nil)
                | split(x::y::tail) =
                      let val (p, q) = split(tail)
                       in (x::p, y::q)
                      end
            val (p, q) = split(x)
         in merge (sort less p, sort less q)
        end
```

This function does indeed work as expected, and its type is

```
    ('a * 'a -> bool) -> 'a list -> 'a list
```

If we had written this entire function before testing it, and made the same mistake by leaving out the line saying

```
      | sort less [a] = [a]
```

then the compiler would have reported the type as

```
    ('a * 'a -> bool) -> 'b list -> 'a list
```

thus showing again that something was wrong.

Reflections

An ML function computes a result from its arguments and whatever other information is available to it at the time it is called. Thus it cannot create a result of a completely new type; the result type must somehow depend on the types that already exist.

That means that if the compiler determines the type of a function to be, say,

```
    'a list -> 'b list
```

one should be instantly suspicious. Where could 'b have come from? It's not the type of something that already exists; such a type would be known explicitly and would not have to be expressed in terms of the type metavariable 'b. I can think of only three possibilities:

— The function returns an empty list, so that the types of the elements that the list doesn't contain are irrelevant. One example is the function

```
    fun f(x) = if null(x) then nil else nil
```

— the function raises an exception, so that the type of the value it doesn't return is irrelevant, or

— the function loops and never terminates at all.

Of course, the function could also do one or another of these things depending on its argument. For example:

```
fun f(nil) = nil
  | f([a]) = raise x
  | f(y) = f(y)
```

returns `nil` if given a `nil` argument, raises exception `x` if given an argument that is only a single element, and otherwise runs forever. The type of that function is indeed

```
'a list -> 'b list
```

In any event, if an ML compiler determines that a function returns a type that is unrelated to its argument types, that should be cause for suspicion.

We constantly hear that the earlier a defect is found and corrected, the less expensive it is. In software, that is a powerful argument for strong typing: a compiler that does aggressive type-checking can often detect errors long before the program is actually run.

This particular case, however, is new to my experience: never before had I seen a compiler prove non-termination of a program I had written! The reason in this case is clear: the ML type-checker infers the most general type that can be ascribed to a particular function. To do that, it must effectively do flow analysis. Of course, it is impossible to prove non-termination for all non-terminating programs; this makes it doubly impressive that it can be done at all.

References

The definition of Standard ML is called, logically enough, *The Definition of Standard ML* by Milner, Tofte, and Harper, (MIT Press 1990, ISBN 0-262-13255-9).

Only a masochist would attempt to learn ML from the Definition. A much more approachable (but less rigorous) source is *ML for the Working Programmer* by Laurence Paulson (Cambridge University Press 1991, ISBN 0-521-39022-2).

There are many books that describe Lisp. A reasonable place to start is *Lisp*, by Winston and Horn (Addison-Wesley 1989, 0-201-08319-1).

Finally, all the examples shown here were run on the Standard ML of New Jersey implementation, version 0.75. The current successor to that implementation is available free of charge by anonymous FTP from `research.att.com`; look in directory `dist/ml`.

About the author

Andrew Koenig is a Distinguished Member of Technical Staff at AT&T Bell Laboratories, where since 1986 he has worked mostly on C++. He joined Bell Labs in 1977 after completing an MS degree in computer science at Columbia University. Aside from C++, his work has included programming language design and implementation, security, automatic software distribution, online transaction processing, and computer chess. He is the author of more than seventy articles and the book 'C Traps and Pitfalls.' He is the Project Editor of the ISO/ANSI C++ committee and a member of five airline frequent flyer clubs.

libscheme: Scheme as a C Library

Brent W. Benson Jr.
Harris Computer Systems
`Brent.Benson@mail.csd.harris.com`

Abstract

Because of its small size and simplicity, Scheme is often seen as an ideal extension or scripting language. While there are many Scheme implementations available, their interfaces are often complex and can get in the way of using the implementation as part of a larger software product. The **`libscheme`** library makes the Scheme language available as a C library. Its interface is through a single C header file and it is easily extended with new primitive procedures, new primitive types, and new syntax. It is portable to any system that has an ANSI C compiler and to which Hans Boehm's popular conservative garbage collector [1] has been ported. It has been used to build a variety of special purpose data manipulation tools, and as an extension language for an ethernet monitor.

1 Introduction

There is a long tradition of scripting languages in the Unix community, the canonical example being `/bin/sh` [2]. Scripting languages allow the programmer to express programming ideas at a high level, and can also be designed in such a way that the language interpreter can be included as an extension language inside of other programs. When a program provides a powerful extension language to end users, it often increases the utility of the program by orders of magnitude (consider GNU Emacs and GNU Emacs Lisp as an example). While in recent years there has been an explosion of general purpose extension and scripting languages (e.g., Python [6] and Elk [4]), one language has had a dramatic increase in popularity and seems to have become the de facto extension language. That language is Tcl [5].

It is the author's opinion that the popularity of Tcl is primarily due to the ease with which it can be embedded into C applications (its interface is through a single C header file and a standard C library archive file) and the ease with which it can be extended with new primitive commands. The `libscheme` library attempts to learn from Tcl's success by making Scheme [3] available as a C library and by providing simple ways to extend the language with new procedures and syntax. While Scheme is not as convenient as Tcl in the role of an interactive shell program, it has several advantages over Tcl with respect to writing scripts:

1. Lexical Scope
2. Nested procedures
3. A richer set of data types
4. Extensible syntax

In addition, `libscheme` allows the application writer to extend the interpreter with new data types that have the same standing as built in types. It also provides a conservative garbage collector that can be used by application and extension writers.

2 Scheme

Scheme is a small, lexically scoped dialect of Lisp that is based on the principle that a programming language should not include everything but the kitchen sink, but rather it should provide a framework in which it is easy to build the kitchen sink.

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today. [3]

These properties make Scheme a good general purpose programming language and also an ideal extension language. Its conceptual simplicity allows it to be implemented with a relatively small number of lines of code, while providing powerful high level programming constructs such as procedures that can be nested and used as first class values, and high-level data structures like lists, vectors and strings. It also removes the burden of memory management from the programmer, usually through some sort of garbage collector.

Scheme supports all major programming paradigms in use today including functional, procedural, and object oriented. It scales well from small applications to large software systems.

2.1 An Example Procedure

Let us examine a small Scheme procedure to get a feel for the language. The procedure in figure 1 splits a string of characters into a list of strings based on a delimiter character.

```
(define (split-string string delimiter)
  (let ((len (string-length string)))

    ; Collect characters until we reach a delimiter character,
    ; at which time we extract a field and start looking for
    ; the next delimiter.
    ;
    (define (collect start end)
      (cond
       ((= end len)
        (list (substring string start end)))
       ((char=? (string-ref string end) delimiter)
        (cons (substring string start end)
              (collect (+ end 1) (+ end 1))))
       (else (collect start (+ end 1)))))

    ; Start at the beginning of the string.
    ;
    (collect 0 0)))
```

Figure 1: A sample Scheme function

In this example we have a top-level definition of the `split-string` function and an *internal definition* of the `collect` function. An internal, or *nested* function like `collect` has access to all lexical variables in the scope of its definition. More specifically, `collect` has access to the lexical variables `string`, `delimiter` and `len`. The `let` special form establishes a series of local variables and initial bindings. In split-string the let establishes only the binding of len. The cond special form evaluates the test in each of its clauses, performing the clauses action(s) when it finds the first test to succeed. The else clause of a cond form is executed if no other clause succeeds. Scheme also has other standard control constructs like if and case.

```
> (split-string "brent:WgG6SfAUnX51Q:5359:100:Brent Benson" #\:)
("brent" "WgG6SfAUnX51Q" "5359" "100" "Brent Benson")
>
```

2.2 Closures

Procedures are first class in Scheme, meaning that procedures can be assigned to variables, passed as arguments to other procedures, be returned from procedures, and be applied to arbitrary arguments. When they are created, procedures "capture" variables that are free in their bodies.[1] Because of this "closing over," procedures are also known as *closures* in Scheme.

Closures can be used for a wide variety of purposes. They can be used to represent objects in an object-oriented framework, to model actors, to represent partially computed solutions, etc.

Unnamed procedures are created with the lambda special form. The existence of unnamed procedures frees the programmer from making up names for simple helper functions. For example, many Scheme implementations have a sort procedure that accepts a list or vector of elements, and a comparison procedure that is used to establish an ordering on the elements. It is often useful to use an unnamed procedure as the comparison function:

```
> (sort '(1 6 3 4) (lambda (n1 n2) (< n1 n2)))
(1 3 4 6)
> (sort #("jim" "brent" "jason" "todd")
        (lambda (s1 s2) (string<? s1 s2)))
#("brent" "jason" "jim" "todd")
```

Note that (1 2 3 4) is a list of numbers and #("jim" ...) is a vector of strings.

The next example shows the definition of a procedure make-counter that returns another procedure created with lambda that closes over the count variable.

```
> (define (make-counter)
    (let ((count 0))
      (lambda ()
        (set! count (+ count 1))
        count)))
make-counter
> (define c1 (make-counter))
c1
> (c1)
1
> (c1)
2
```

Since no one else has access to count it becomes private to the closure and can be modified and used within its context. In this particular case, count is incremented and its value returned when the procedure returned from make-counter is called.

2.3 Syntax

As you have probably already noticed, Scheme's syntax is Lisp-like. All function applications are in fully-parenthesized prefix form. While some find this sort of syntax unwieldy, it has the

[1]A variable is free in a procedure if it is not contained in the parameter list.

advantage that Scheme forms are actually lists which can be easily manipulated with standard list primitives. The libscheme library supports the defmacro special form that can be used by end users to create new special forms. A special form is a form that is evaluated by special rules. For example, the if special form only evaluates its "then" condition if its test expression evaluates to true, otherwise it evalutes its "else" expression.

These macros are much more powerful than the simple token-based substitution macros provided by languages like C.

3 An Application that Uses libdwarf

The libscheme library makes Scheme available as a C library. Its interface is through a single C header file. Scheme expressions can be read from an arbitrary input stream, evaluated with respect to a particular environment, and printed to an arbitrary output stream. The user can extend the interpreter by adding bindings to the global environment. Each binding can provide a new primitive written in C, a new syntax form, a new type, a constant, etc.

3.1 An Example

DWARF is a full-featured and complex debugging information format [7]. Our example program, dwarfscheme, is an interface that allows the user to browse DWARF information in an object file by providing stubs to the libdwarf [8] library. Figure 2 shows a sample dwarfscheme dialogue.

In this example the user invokes dwarfscheme, opens the file "a.out" for DWARF reading, defines a function for printing out debugging information entries (DIEs), and prints out the first DIE. This example shows how dwarfscheme would be used by an end user. Next, we will examine the way that the dwarfscheme executable was created using libscheme and the libdwarf libraries.

The program dwarfscheme is an executable that was produced by linking libscheme with a set of DWARF manipulating primitives, a read-eval-print loop that initializes the primitives, and the libdwarf library that is provided as a system library. The main routine for dwarfscheme appears in figure 3.

This main routine is a boiler-plate routine that is used when the application writer wants to make the application a Scheme read-eval-print loop. The thing that differentiates the main routines in different applications is the initializations that are done on the environment. In this case, we create a basic environment containing the standard libscheme bindings, and then add the DWARF specific bindings to it by calling init_dwarf(). The rest of the routine takes care of the business of establishing an error handler, printing out a prompt, reading an expression, evaluating the expression, and printing out the result.

The application writer can also embed libscheme in an application that is not structured as a read-eval-print loop. For example, at program startup a windowing application might initialize a libscheme environment, read and evaluate Scheme expressions representing configuration information from a user configuration file, and then enter its event loop. The user might bring up a dialog box in which she can evaluate Scheme expressions to further configure and query the system's state.

The major part of the DWARF initialization routine, init_dwarf() appears in figure 4. It consists of calls to scheme_make_type() to establish new data types, and then several calls to scheme_add_global() to add new global bindings to the environment provided as an argument. Each call to scheme_add_global() provides the Scheme name for the global, the initial value for the variable (in this case a new primitive that points to its C implementation), and an environment to which the global should be added. All routines and variables that are part of the dwarfscheme interface begin with a dw prefix, while routines and variables from the system-supplied libdwarf library begin with a dwarf prefix.

This practice of calling an initialization routine with the environment for each logical piece of code is only a convention, but is a helpful way of organizing libscheme code. The libscheme library itself is organized this way. Each file contains an initialization function that establishes that file's primitives.

```
$ ./dwarfscheme
> (define dbg (dwarf-init "a.out"))
dbg
> (define die1 (dwarf-first-die dbg))
die1
> (define (dwarf-print-die die)
    (display (dwarf-tag-string (dwarf-tag die)))
    (newline)
    (for-each
      (lambda (attr)
        (display "  ")
        (display (dwarf-attr-name-string (dwarf-attr-name attr)))
        (display " ")
        (write (dwarf-attr-value attr))
        (newline))
      (dwarf-attributes die)))
dwarf-print-die
> (dwarf-print-die die1)
DW_TAG_compile_unit
  DW_AT_comp_dir "CX/UX:/jade2/ccg/brent/misc/package/dwarf"
  DW_AT_identifier_case 0
  DW_AT_low_pc 198304
  DW_AT_high_pc 198344
  DW_AT_language 2
  DW_AT_name "t.c"
  DW_AT_producer "Harris C Compiler - Version build_c_7.1p1_03"
  DW_AT_stmt_list 0
#t
> (exit)
$
```

Figure 2: `dwarfscheme` dialogue

```
#include <stdio.h>
#include "scheme.h"
#include "dwarfscheme.h"

main()
{
  Scheme_Env *env;
  Scheme_Object *obj;

  env = scheme_basic_env ();
  init_dwarf (env);
  scheme_default_handler ();
  do
    {
      printf ("> ");
      obj = scheme_read (stdin);
      if (obj == scheme_eof)
        {
          printf ("\n; done\n");
          exit (0);
        }
      obj = scheme_eval (obj, env);
      scheme_write (stdout, obj);
      printf ("\n");
    }
  while ( 1 );
}
```

Figure 3: `dwarfscheme` read-eval-print loop

```
static Scheme_Object *dw_debug_type;
static Scheme_Object *dw_die_type;
static Scheme_Object *dw_first_die (int argc, Scheme_Object *argv[]);
...
void
init_dw (Scheme_Env *env)
{
  dw_debug_type = scheme_make_type ("<debug>");
  dw_die_type = scheme_make_type ("<die>");
  dw_attribute_type = scheme_make_type ("<attribute>");
  scheme_add_global ("dwarf-init",
                     scheme_make_prim (dw_init), env);
  scheme_add_global ("dwarf-first-die",
                     scheme_make_prim (dw_first_die), env);
  scheme_add_global ("dwarf-next-die",
                     scheme_make_prim (dw_next_die), env);
  scheme_add_global ("dwarf-tag",
                     scheme_make_prim (dw_tag), env);
  ...
}
```

Figure 4: The DWARF primitive initialization routine

A sample primitive is shown in figure 5. Each libscheme primitive accepts an argument count and a vector of evaluated arguments. Each primitive procedure is responsible for checking the number and type of its arguments. All Scheme objects are represented by the C type Scheme_Object (see section 4.1). The types Dwarf_Debug and Dwarf_Die are foreign to libscheme and are provided by the libdwarf library.

```
static Scheme_Object *
dw_first_die (int argc, Scheme_Object *argv[])
{
  Dwarf_Debug dbg;
  Dwarf_Die die;

  SCHEME_ASSERT ((argc == 1),
                 "dwarf-first-die: wrong number of args");
  SCHEME_ASSERT (DWARF_DEBUGP (argv[0]),
                 "dwarf-first-die: arg must be a debug object");
  dbg = (Dwarf_Debug) SCHEME_PTR_VAL (argv[0]);
  die = dwarf_nextdie (dbg, NULL, NULL);
  if (! die)
    {
      return (scheme_false);
    }
  else
    {
      return (dw_make_die (die));
    }
}
```

Figure 5: A dwarfscheme primitive

The SCHEME_ASSERT() macro asserts that a particular form evaluates to true, and signals an error otherwise. The dw_first_die() routine first checks for the correct number of arguments, then it checks that the first argument is an object with type dw_debug_type. Next, it extracts the pointer value representing the DWARF information in the file from the first argument, a Scheme_Object. It then calls a libdwarf function, dwarf_nextdie() and returns an appropriate value—a new dw_die_type object if there is another DIE, the Scheme false value otherwise. The dw_make_die() routine accepts a Dwarf_Die as an argument and returns a libscheme object of type dw_die_type that contains a pointer to the Dwarf_Die structure.

Now that we have a feel for the way that libscheme is extended, we will take a closer look at the design of libscheme itself.

4 libscheme Architecture

This section describes some specifics of libscheme's implementation. An important feature of its design is that beyond a small kernel of routines for memory management, error handling, and evaluation, all of its Scheme primitives are implemented in the same way as non-libscheme extensions. This is similar to Tcl's implementation strategy.

4.1 Object Representation

Every object in libscheme is an instance of the C structure Scheme_Object. Each instance of Scheme_Object contains a pointer to a type object (that also happens to be a Scheme_Object) and two data words. If an object requires more than two words of storage or if the object is some other type of foreign C structure, it is stored in a separate memory location and pointed to by the `ptr_val` field. The actual definition of `Scheme_Object` appears in figure 6.

```
struct Scheme_Object
{
  struct Scheme_Object *type;
  union
    {
      char char_val;
      int int_val;
      double double_val;
      char *string_val;
      void *ptr_val;
      struct Scheme_Object *(*prim_val)
        (int argc, struct Scheme_Object *argv[]);
      struct Scheme_Object *(*syntax_val)
        (struct Scheme_Object *form, struct Scheme_Env *env);
      struct { struct Scheme_Object *car, *cdr; } pair_val;
      struct { int size; struct Scheme_Object **els; }
               vector_val;
      struct { struct Scheme_Env *env;
               struct Scheme_Object *code; } closure_val;
    } u;
};
```

Figure 6: The definition of `Scheme_Object`

While many Scheme implementations choose to represent certain special objects as immediate values (e.g., small integers, characters, the empty list, etc.) this approach was not used in `libscheme` because the "everything is an object with a tag approach" is simpler and easier to debug. A side effect of this decision is that `libscheme` code that does heavy small integer arithmetic will allocate garbage that must be collected, in contrast to higher performance Scheme implementations that only dynamically allocate very large integers.

4.2 Primitive Functions

Primitive functions in Scheme are implemented as C functions that take two arguments, an argument count and a vector of `Scheme_Objects`. Each primitive is responsible for checking for the correct number of arguments—allowing maximum flexibility for procedures of variable arity—and for checking the types of its arguments. All arguments to a primitive function are evaluated before they are passed to the primitive, following Scheme semantics. If the application writer wants to create a primitive that doesn't evaluate its arguments, she must use a syntax primitive. C functions are turned into `libscheme` primitives with the `scheme_make_prim()` function that accepts the C function as an argument and returns a new Scheme object of type `scheme_prim_type`.

4.3 Primitive Syntax

The user can add new primitive syntax and special forms to `libscheme`. A `libscheme` syntax form is implemented as a C function that takes two arguments, an expression and an environment in which the expression should be evaluated. The form is passed directly to the syntax form with no evaluation performed. This allows the syntax primitive itself to evaluate parts of the form as needed. Figure 7 shows the implementation of the `if` special form. Note that `if` cannot be implemented as a procedure because it must not evaluate all of its arguments. The `scheme_eval()` function evaluates a `libscheme` expression with respect to a particular environment.

```
static Scheme_Object *
if_syntax (Scheme_Object *form, Scheme_Env *env)
{
  int len;
  Scheme_Object *test, *thenp, *elsep;

  len = scheme_list_length (form);
  SCHEME_ASSERT (((len == 3) || (len == 4)),
                 "badly formed if statement");
  test = SCHEME_SECOND (form));
  test = scheme_eval (test, env);
  if (test != scheme_false)
    {
      thenp = SCHEME_THIRD (form);
      return (scheme_eval (thenp, env));
    }
  else
    {
      if (len == 4)
        {
          elsep = SCHEME_FOURTH (form);
          return (scheme_eval (elsep, env));
        }
      else
        {
          return (scheme_false);
        }
    }
}
```

Figure 7: The `if` special form

C functions that represent syntax forms are turned into Scheme objects by passing them to the `scheme_make_syntax()` procedure which returns a new object of type `scheme_syntax_type`.

4.4 Type Extensions

Scheme as defined in its standard has the following data types: boolean, list, symbol, number, character, string, vector, and procedure. While Scheme in its current form does not allow the creation of user-defined types, the `libscheme` system allows users to extend the type system with new types by calling the `scheme_make_type()` function with a string representing the name of the new type. This function returns a type object that can be used as a type tag in subsequently created objects. Normally, types are created in a file's initialization function and objects of the new type are created using a user-defined constructor function that allocates and initializes instances of the type.

In figure 8 we see the constructor for the `dw_debug_type` type from our `dwarfscheme` example. It accepts an object of type `Dwarf_Debug`, a pointer to a C structure defined in the `libdwarf` library, allocates a new `Scheme_Object`, sets the object type, and stores the pointer to the foreign structure into the `ptr_val` slot of the object.

```
static Scheme_Object *
dw_make_debug (Dwarf_Debug dbg)
{
  Scheme_Object *debug;

  debug = scheme_alloc_object ();
  SCHEME_TYPE (debug) = dw_debug_type;
  SCHEME_PTR_VAL (debug) = dbg;
  return (debug);
}
```

Figure 8: An object constructor

It is often convenient to define a macro that checks whether a libscheme object is of a specified type. The macro defined in dwarfscheme for the DWARF debug object looks like this:

```
#define DW_DEBUGP(obj) (SCHEME_TYPE(obj) == dwarf_debug_type)
```

The 'P' at the end of DW_DEBUGP indicates that the macro is a predicate that returns a true or false value. All of the builtin types have type predicate macros of this form (e.g., SCHEME_PAIRP, SCHEME_VECTORP, etc.).

4.5 Environment Representation

The state of the interpreter is contained in an object of type Scheme_Env. The environment contains both global and local bindings. The definition of the Scheme_Env structure is shown in figure 9. The global variable bindings are held in a hash table. The local bindings are represented by a vector of variables (symbols) and a vector of corresponding values. An environment that holds local variables points to the enclosing environment with its next field. Therefore, variable value lookup consists of walking the environment chain, looking for a local variable of the correct name. If no local binding is found, the variable is looked for in the global hash table.

```
struct Scheme_Env
{
  int num_bindings;
  struct Scheme_Object **symbols;
  struct Scheme_Object **values;
  Scheme_Hash_Table *globals;
  struct Scheme_Env *next;
};
```

Figure 9: The Scheme_Env structure

Table 1 lists the environment manipulation functions. Unless the user is adding special forms that create variable bindings, she usually only needs to worry about the scheme_basic_env() and scheme_add_global() functions. The scheme_basic_env() function is used to create a new environment with the standard Scheme bindings which can then be extended with new primitives, types, etc. using scheme_add_global().

4.6 Interpreter Interface

The libscheme functions that are used for reading, evaluating and writing expressions are listed in table 2.

`scheme_basic_env ()`	Return a new `libscheme` env
`scheme_add_global (name, val, env)`	Add a new global binding
`scheme_add_frame (syms, vals, env)`	Add a frame of local bindings
`scheme_pop_frame (env)`	Pop a local frame
`scheme_lookup_value (sym, env)`	Lookup the value of a variable
`scheme_lookup_global (sym, env)`	Lookup the value of a global
`scheme_set_value (sym, val, env)`	Set the value of a variable

Table 1: Environment manipulation functions

`scheme_read (fp)`	Read an expression from stream
`scheme_eval (obj, env)`	Evaluate an object in environment
`scheme_write (fp, obj)`	Write object in machine readable form
`scheme_display (fp, obj)`	Write object in human readable form

Table 2: Interpreter functions

These functions can be used in the context of a read-eval-print loop or called at arbitrary times during program execution.

4.7 Error Handling

The `libscheme` library provides rudimentary error handling support. Errors are signaled using the `scheme_signal_error()` function, or by failing the assertion in a `SCHEME_ASSERT()` expression. If the default error handler is installed by calling `scheme_default_handler()`, then all uncaught errors will print the error message and `abort()`. Errors can be caught by evaluating an expression within a `SCHEME_CATCH_ERROR()` form. This macro evaluates its first argument. If an error occurs during the execution, the value second argument is returned, otherwise, the value of the first expression is returned.

```
obj = scheme_read (stdin);
result = SCHEME_CATCH_ERROR (scheme_eval (obj, env), 0);
if (result == 0)
   {
     /* error handling code */
   }
else
   {
     scheme_write (stdout, result);
   }
```

4.8 Memory Allocation/Garbage Collection

The `libscheme` library uses Hans Boehm and Alan Demers' conservative garbage collector [1]. It provides a replacement for the C library function `malloc()` called `GC_malloc()`. The storage that is allocated by `GC_malloc()` is subject to garbage collection. The collector uses a conservative approach, meaning that when it scans for accessible objects it treats anything that could possibly point into the garbage collected heap as an accessible pointer. Therefore, it is possible that an integer or a floating point number that looks like a pointer into this area could be treated as a root and the storage that it points to would not be collected. In practice, this rarely happens.

Users of `libscheme` can use the garbage collector in their own program and are strongly encouraged to make use of it when extending `libscheme`. Normally the user can simply call `scheme_alloc_object()` to allocate a new `Scheme_Object`, but occasionally other types of objects need to be allocated dynamically.

The Boehm/Demers collector is freely available and can run on most any 32-bit Unix machine.

5 Pros, Cons and Future Work

The `libscheme` library is simple to understand and use. It builds on the powerful semantic base of the Scheme programming language. The library also provides several powerful extensions to Scheme including an extensible type system and user defined structure types.

The `libscheme` interpreter is not very fast. The primary reason is an inefficient function calling sequence that dynamically allocates storage, creating unnecessary garbage. This issue is being addressed and future versions should be a great deal more efficient. In any case, `libscheme` is intended primarily for interactive and scripting use for which its performance is already adequate.

When compared to a language like Tcl, Scheme is not as well suited for interactive command processing. A possible solution is to add a syntax veneer on top of the parenthetical Scheme syntax for interactive use. On the other hand, Scheme's clean and powerful semantics provide a sizeable advantage over Tcl for writing large pieces of software. It also has the advantage of real data types rather than Tcl's lowest common denominator "everything is a string" approach.

The `libscheme` library has already been used in many small projects. The author plans to make `libscheme` even more useful by providing a variety of useful bindings including interfaces to the POSIX system calls, a socket library, a regular expression package, etc.

6 Conclusion

The `libscheme` library makes Scheme available as a standard C library and is easily extended with new primitives, types, and syntax. It runs on most Unix workstations including Harris Nighthawks, Suns, Intel 386/486 under Linux and OS/2, HP9000s, DECstations and DEC Alphas running OSF/1, IBM RS/6000s, NeXT machines and many others. Its simplicity and extensibility lends itself to use as an extension or scripting language for large systems. Currently it is being used by the DNPAP team at Delf University of Technology in the Netherlands as part of their ethernet monitor, and is being evaluated for use in a variety of other projects. The latest version of `libscheme` can be obtained from the Scheme Repository, `ftp.cs.indiana.edu`, in the directory `/pub/imp/`.

References

[1] Hans Boehm and M. Weiser. *Garbage Collection in an Uncooperative Environment*. Software Practice and Experience. pp. 807-820. September, 1988.

[2] Stephen Bourne. *An Introduction to the UNIX Shell*. Berkeley 4.3 UNIX User's Supplementary Documents. USENIX Association.

[3] William Clinger and Jonathan Rees (Editors). *Revised^4 Report on the Algorithmic Language Scheme*. Available by anonymous ftp from `altdorf.ai.mit.edu`. 1991.

[4] Oliver Laumann. *Reference Manual for the Elk Extension Language Kit*. Available by anonymous ftp from `tub.cs.tu-berlin.de`.

[5] John Ousterhout. *Tcl: an Embeddable Command Language*. Proceedings of the Winter 1990 USENIX Conference. USENIX Association. 1990.

[6] Guido van Rossum. *Python Reference Manual*. Release 1.0.2. Available by anonymous ftp from `ftp.cwi.nl`. 1994.

[7] *DWARF Debugging Information Format*. Unix International. Available by anonymous ftp from `dg-rtp.dg.com`. 1994.

[8] *DWARF Access Library (libdwarf)*. Unix International. 1994.

The Author

Brent Benson received a BA in Mathematics from the University of Rochester 1990 and completed the work for his MS in Computer Science at the University of New Hampshire in 1992. He has been a senior software engineer in the small but feisty compiler group at Harris Computer Systems since 1992.

A New Architecture for the Implementation of Scripting Languages

Adam Sah and Jon Blow
Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
{asah, blojo}@cs.Berkeley.EDU

September 12, 1994

Abstract

Nearly all scripting languages today are implemented as interpreters written in C. We propose an alternate architecture where the language is translated into the dynamic language Scheme [R4RS]. The plethora of high quality, public domain Scheme implementations give the developer a wide selection of interpreters, byte compilers, and machine code compilers to use as targets for her VHLL.

Our VHLL, Rush, provides high-level features such as automatic type conversion and production rules [SHH86][Ston93]. Performance benchmarks show that our system runs with acceptable speed; in fact, Rush programs run much more quickly than their equivalents in languages such as Tcl and Perl4. Whereas those languages are coded in C, Rush takes advantage of Scheme's existing high-level features, saving development time. Since the features provided by Scheme are among those most VHLLs share, we expect this approach to be widely applicable.

1 Introduction

We propose that VHLLs be implemented as source-to-source translators emitting a high-level language such as Scheme [R4RS] and which use publicly available implementations of the target language to execute the program. In other words, we advocate implementing VHLLs that use Scheme as their assembly language. Though others have indeed suggested that Scheme has properties useful as a compiler representation language, such observations and proposals have rarely gone beyond speculation. We advocate the approach because we have tried it and it works.

We think of VHLLs as supporting some of these features:

- Automatic memory management.
- Comfortable syntax when interpreted from a command line.
- Primitive support for complex data structures.
- Easy translation of data to string form and back.

*supported in part by a grant from the National Science Foundation #NSF-IRI9107455

- A large and useful standard procedure library.
- Ability to define procedures at runtime.
- ...other "high level" features like variable traces.

Implementing any single high level feature is not difficult, but putting them together can be challenging. Most VHLLs in use today, such as Tcl and Perl, incur large performance penalties to provide their features. Scheme, on the other hand, provides many of these features with implementations that are fairly efficient and interact cleanly; by using Scheme as a back-end the VHLL designer can adopt desired features with minimal effort. This allows him to concentrate development time on the features that matter most, the ones specific to the VHLL being implemented, rather than reinventing old wheels.

We designed a VHLL called Rush and implemented it as a Scheme-generating compiler. This paper explains how we arrived at the decision to use Scheme as an intermediate form (section 2), lists some of the problems that we encountered using Scheme (section 3), and shows how some high level Rush features map elegantly into Scheme, where such a mapping would be far more difficult into other languages (section 4). In section 5 we present a small performance analysis of Rush, showing that it executes code far more quickly than similar languages such as Tcl and Perl4, even though our system wasn't hand-tuned. In section 6 we examine the source trees for Perl4, Tcl7.3, Rush, and SCM4d3 (a public domain Scheme interpreter) in an attempt to pinpoint the work we saved by using Scheme as our runtime environment.

2 Scheme as an Intermediate Representation

2.1 What is an IR?

An Intermediate Representation (IR) is a language that lies between the parsing module of a compiler and the target-code generator. Source code in the high level language is compiled into the IR, and then the IR code is either compiled into machine code or interpreted on the fly. The benefits of IRs are numerous. First, an IR makes the source language more portable, since the IR$\rightarrow$machine code back-end can be retargeted without changing the front end.

As an example, the GNU C Compiler (gcc) converts C source code into an intermediate representation called RTL ("register transfer language"). RTL is a very low level representation, almost a generic assembly language. The compiler analyzes and optimizes the RTL code, then uses one of many back-end programs to convert RTL into machine-specific assembly language. To retarget gcc, one need only write a new RTL translator; to adapt gcc to compile a new language, one need only write a translator from that language to RTL.

A careful selection of the IR can make analysis and optimization easier; this is widely believed to be the case with three-address codes like Single Static Assignment [RZ88] [CFRWZ91] (for Pascal-like languages) and Continuation-Passing Style [Shiv88] [App92] for Lisp-like languages. As we discuss in sections 3 and 4, analysis can be very important when designing and implementing VHLLs; it is a fallacy to assume that perforance doesn't matter. The choice of IR directs how analysis will be performed, and to what degree existing implementations of the IR already perform the desired analyses.

Problems can arise from a bad choice of IR. A poor matchup between the source language and the IR will require the language implementor to expend a lot of effort bridging semantic gaps. For example, translating Scheme into C is frustrating because of the need to support call-with-current-continuation (call/cc) and tail-recursion elimination, both of which have unpleasant interactions with memory allocation and C's stack-based procedure frame model, as implemented in virtually every C compiler today[Bart89].

If the IR you select has already been implemented, you save the implementation time for the back end. As an example of this, it has become common practice to write compilers that emit C code which is then compiled into an executable, rather than emitting machine code directly. (The INTERCAL-C Compiler [WL73] is such a program.) At the very least this approach saves the language implementor from having to write register allocation algorithms, some of the most complex code in modern optimizing compilers.

2.2 An Introduction to Scheme

Throughout this paper we will discuss the Scheme output code of our compiler system. We cannot provide here a full description of Scheme; for that we refer the reader to [SICP] and [SF89]. However, to give a taste of the way Scheme works, we present a short description of the behavior of Scheme's procedures in comparison to C's. Those familiar with Scheme may skip this section with impunity.

Procedure calls in Scheme are written as nested lists; each list is a series of expressions, the first denoting the procedure to call and the rest listing the arguments. For example:

```
(foo 4 (bar 3))
```

calls the procedure `foo` with two arguments: `4` and the result of calling `bar` with `3`. This is equivalent to the C expression:

```
foo(4, bar(3))
```

Variables in Scheme can be defined with the keyword `define`. Procedures can be created with the constructor `lambda`, which takes as parameters a list of arguments and an expression to act as the procedure body. Here is the definition of a "square" function:

```
(define square (lambda (x) (* x x)))
```

This is equivalent to the C function:

```
int square(int x) { return x*x; }
```

Procedures in Scheme can be nested. Scheme is lexically scoped. By combining lexical scoping and use of `lambda` we can create objects with state in a simple way, like so:

```
(define make-adder
   (lambda (x)
      (lambda (y) (+ x y))))
```

This means that `make-adder` is a function of one argument `x` that, when called, returns another procedure (using "`lambda`") that takes one argument `y` and adds it to `x`. Here is an example of its use:

```
(define adder1 (make-adder 6))          ; adder1 is now a function
(adder1 5)                              ; returns 11

(define adder2 (make-adder 10))
(adder2 6)                              ; returns 16
(adder1 (adder2 10))                    ; returns 26
```

Each procedure has its own value of `x` which it remembers for as long as it exists. Such state values could even be modified. Using "`lambda`" this way is perfectly natural in Scheme.

2.3 Why Scheme?

Before espousing the benefits of Scheme, we should make it clear that Scheme is not a panacea; neither is it the only logical choice as a code generation language for VHLLs. Some discussion of other possible languages is presented at the end of this paper.

Scheme's structure is simple, consisting of command/argument lists enclosed in parentheses. This syntax has led many critics to complain that the language is unreadable; however, for machine-generated code readability is not so important, and syntactic simplicity is a boon since it means that the code-generation routines of the VHLL compiler can also be simple.

In the segments below we discuss our reasons for using Scheme; first we talk about the availability and versatility of Scheme implementations, then we discuss at length the semantic advantages that Scheme provides.

2.4 Current Scheme Implementations

Many Scheme implementations are available, both commercially (as Chez Scheme) and as freeware on the Internet (eg. SCM, Scheme→C, Bigloo, MIT-Scheme, schemelib, etc.). More importantly, these are not weekend hacks or student projects; they are well-documented, portable systems. This provides a wide selection of targets to choose from.

2.4.1 Scheme Speed

Scheme can be implemented efficiently [VP89] and this is proven in practice by the above systems. By comparison, using other high-level languages (such as, say, Tcl) might be a bad idea since such languages' semantics guarantee that they will be untowardly difficult to optimize.

Deciding which languages would make suitable back-ends requires one to set a goal about how fast the VHLL should be. In terms of Rush, we measured speed relative to C, deciding that Rush programs should run no more slowly than 10x-50x their C counterparts. In contrast, Tcl is a factor of 1000x-10000x away from C, and pathologic cases (such as array references) even have different computational complexities [Sah94]. In setting a speed goal, we accept that the VHLL we're designing could not compete with C or C++ for speed, but we also insist that it mustn't turn a DEC Alpha into an Atari 800.

2.4.2 Scheme Architecture

Basically all implementations of Scheme are centered around a read-eval-print loop: a routine that reads a string of text, evaluates it as Scheme, and translates the result into a string. The VHLL implementor can use this mechanism directly, parsing the code from her language into Scheme and feeding that Scheme to the read-eval-print routine.

The demo version of the Rush interpreter was a parser connected to a Scheme interpreter via a Unix pipe. This was conceptually dirty but it meant that we could concentrate on writing a correct yacc script to parse Rush and a code generation module to emit valid Scheme. This mechanism took a few hours to write, debug, and test; it's a negligible amount of code. It has since been discarded; now we link the Scheme interpreter as a shared library and call the read-eval-print loop as a procedure from C.

2.5 Advantageous Scheme Semantics

2.5.1 Scoping

Scheme is lexically scoped, which means that a variable is defined and accessible in a program only at the level of nesting where its definition is written, and in routines contained within that definition.

Lexical scoping is generally fast (compared to dynamic scoping, which is used in languages like Tcl and some older versions of Lisp) [ASU86].

2.5.2 First-class Procedures

We described first-class procedures in a previous section introducing Scheme. In Rush first class functions are used to provide many important features. More details about those features can be found in section 4.

2.5.3 Continuations

A continuation is a procedure that represents the of "the rest of the program"— what computations would follow from the current state of program execution. In Scheme, continuations are captured by using the procedure `call-with-current-continuation` (abbreviated `call/cc`). *call/cc* takes a procedure as an argument; it invokes that procedure, passing the current continuation as an argument. Implementationwise, continuations contain the place in the program that execution has reached as well as the state of all active environments (in other words, the values of just about every live variable, except perhaps for globals).

In practice, continuations can be used to implement coroutines, exception handling, and asynchronous event handling, as well as simple control flow (like "goto"). Cooperative multithreading becomes trivial with use of continuations; a context-switch is just a `call/cc` to the next thread on the run queue.

2.5.4 Powerful Macros

One commonly available Scheme macro system is Expansion Passing Style (XPS) [DFH86]. XPS is Turing-complete; it is more expressively powerful than the C preprocessor or even the m4 macro expander. This power gives the user an increased choice of where to compile a language construct. For example, one could write one's entire VHLL as a parser that desugars the VHLL code into parenthesized forms (a pidgin Scheme code), and then use the macro expansion facility to convert this into real Scheme code, with the macros performing all transformations and analyses. From experience, XPS is not hard to use in this way, though we did not use it so heavily for the Rush parser. Instead we emit terse code that is easy to read, using macros to expand forms that would otherwise obfuscate the code.

2.5.5 List-Handling Primitives and Varargs

Linked lists are a fundamental data structure in Scheme. Most common list-manipulation operations are provided as primitives. Some more powerful Scheme procedures take advantage of first-class functions; one example is `map`, which applies a function to every member of a list and returns the resulting list.

Scheme's varargs facility is more versatile than C's; when a varargs procedure is called, Scheme creates a list that contains the values of all the arguments passed, like so:

```
; varargs in scheme:
; myfun requires two arguments, but can take any number beyond that.

(define myfun (lambda (arg1 arg2 . varargs) (map square varargs))

(myfun 1 2 3 4 5)            ; returns (9 16 25)
```

Compare this ease-of-use to C where one must use special accessor macros to get arguments one at a time, knowing the types of each beforehand. An object-level facility has been written for C to make varargs easier; it's called AVCALL, but it isn't universally available [BT93]. Also, it is nearly impossible to use C's native varargs facility to provide varargs as a feature for a VHLL, so the programmer coding the interpreter in C must come up with her own facility.

2.5.6 Garbage Collection

Every Scheme implementation contains a garbage collector. This can be a great benefit. Almost every VHLL provides some form of automatic memory management so that the user does not have to concern himself with explicitly allocating and freeing memory. The authors have yet to hear the benefits of automatic memory management questioned by programmers who have used it in production settings. The single valid complaint is that auto memory management can be slow, but the 5-20% of runtime that it typically consumes for "fast" languages drops well into the noise when the slowness of current VHLLs is considered.

Automatic memory management can be effected in one of two ways. The first (and simplest) way is that the VHLL can be designed with invariants on its data structures such that memory can be reclaimed trivially. The second is that can allocate memory from a garbage collected heap. Actually, current research suggests a third alternative where analysis is used to eliminate the need for garbage collection, but this analyzer will be even more complicated to write than a collector, and is still unproven in practice [TT94].

The first (and worst) approach is used by most VHLLs being written in C; these VHLLs are generally seeking *some* form of memory management, but do not try to implement completely automatic management, since that is more difficult. Tcl, for example, limts its available data structures to those representable by strings; by not offering the notion of a data reference in the language, the Tcl interpreter can easily tell when a value will no longer be used, freeing its memory. The trouble with this approach is twofold. First, it limits the sorts of data structures the language can have and how they can be implemented. In the case of Tcl, this necessitates some copying of data as part of its pass-by-value calling convention. The second problem is that this approach requires the C programmer to maintain strange invariants during callouts from the VHLL. The same problem arises when using a garbage collector, although the selection of a C-compatible garbage collector such as Boehm-Weiser [BW88] might eliminate this need.

2.5.7 Smarter Arithmetic

Scheme makes a tradeoff of speed-for-features in its arithmetic library, but the features provided are very useful.

First, Scheme provides integers of unbounded size ("bignums"), so arithmetic overflow simply does not happen. Most Scheme implementations will cause integers to flow smoothly between several representations depending on their size, all invisibly to the user; an integer less than 2^{31} might be stored in the same form as a C integer, but a larger number would be stored in bignum format.

Scheme also provides precise notions of real and rational numbers, rather than C's floating-point approximations. Again, this has the effect of shielding the casual user from bizarre errors arising from simple math over- and under- flow.

The strength of Scheme's arithmetic handling can be useful in maintaining consistent language semantics. Here we have an example from Tcl 7.3, where evaluating the command:

```
[expr 1234567890*1234567890]
```

produces different results on different systems: on an Alpha running OSF/1 v2.1, the answer is 1524157875019052100, whereas on an Intel 486 running Linux, the answer is 304084036. This is because registers are 64 bits wide on an Alpha but only 32 bits wide on a 486. Perl4 seems to implement integer arithmetic using double-precision floating point, so "print 1524157875019052000-10" results in "1524157875019052000" (which, by the way, is Perl4's answer for the above multiplication.) Since VHLLs generally attempt to abstract the machine from the user, such behavior is in conflict with the spirit of the languages.

Detecting math overflow in C is nontrivial, as is writing the infinite precision math library. To give credit, the Free Software Foundation's "gmp" package is starting to fill this gap for floating point, and Perl4 has already implemented infinite precision integer arithmetic.

2.5.8 SLIB

Aubrey Jaffer's SLIB is a portable, R4RS compliant Scheme library that implements nearly every library feature that's present in VHLLs like Perl4 and Tcl. Without producing a white sheet for it, we will summarize a few of its features:

- String handling and formatting library.
- Data structures: collections, hash tables, queues and priority queues, trees, and records.
- Bitwise manipulation, list routines, random number generation, and sorting.

A third party pattern matching library is also available from `titan.cs.rice.edu:/public/wright/match.tar` As we will explain later, a C callout mechanism is assumed for our VHLL, so additional C libraries should be callable from Scheme. For example, making a Scheme binding for the Tk graphical user interface is reasonably straightforward (and in fact has already been done).

3 Some Downsides to Scheme

Scheme has several flaws from a VHLL standpoint that make it an imperfect code generation language. We believe that these do not outweigh the benefits provided. In this section, we detail the major flaws we encountered during the implementation of Rush.

3.1 C Callouts Made Hard

Providing a callout mechanism to C is important to many VHLL designs; it can be invaluable because of the enormous body of code written in C and the comparative difficulty of getting at this code through other means.

Unfortunately, Scheme has no standard facility for calling foreign functions. Sadly, neither does any other dynamic language we looked at that runs at reasonable speed, so this is a generic problem. Part of our current work is to specify and build this mechanism. It is not difficult to add this to any individual Scheme implementation; our goal would be to target this for a number of Schemes. The major problem is that C programs need to have standard ways to access and build Scheme data structures, and they must do it in such a way that the garbage collector is able to keep track.

The other reasonable option is to provide a string-based callout mechanism, as Tcl does. In that case the information you send to C is converted into a string and passed to C routines which translate the strings into native C structures. This incurs a tremendous performance penalty. (For Tcl 7.3 this is not a problem since all Tcl data is in string form; Tcl has already accepted the performance hit.) This string-passing method only works well for simple data structures; for example, creating a string representation of a closure is very hard [SG90]. In the current version of Rush, we have implemented a string-based callout mechanism, but it does not support closures.

3.2 'goto' Considered Useful

Whereas 'goto' has often been considered harmful, programmers typically demand it in limited forms; the most popular are constructs like "break", "return", "continue", and exception handlers. In Scheme the natural way to alter control flow like this is to create a continuation at the *destination* of the jump and then call that continuation at the source. (This works much like the C procedures `setjmp` and `longjmp`.) However, setting up a continuation can be very slow under some Scheme implementations. On an Alpha using Scheme→C, for example, we were measuring values of 80+ μsec (8000+ instructions) in typical cases. This is because Scheme→C stores call frames on the stack, so `call/cc` must copy the stack into the heap to save a continuation. On systems where `call/cc` is not so slow, call frames are allocated on the heap, causing more frequent garbage collections, slowing down the system in general.

Since we refused to use continuations to implement loop control in Rush, we devised a code manipulation that converts nonlocal gotos into return values from nested scopes. This necessitates a few tricks: First, we require that no code may follow a nonlocal goto statement in any single basic block. Since such code would be unreachable, it deserves a compiler warning anyway.

The second and more challenging trick is to properly capture the results of each code block that *could* result in a nonlocal goto. For example, here is some Rush code:

```
proc foo () {
    for {i=0} {i<100} {i++} {
        if {i>90} {
            break;
        }
        puts "hi"
    }
}
```

In this example, the "break" performs a "goto" to the end of the `for` loop body; in order to implement it, the compiler generates code of this form:

```
...
; then => 'break       else => 'blah
(let ((result (if (> i 90) 'break 'blah)))
   (case result
        ((break) 'break)
        ((else <puts "hi", increment i, recursively call loop>))
)
```

This results in straight-line code becoming nested when multiple statements in a block can perform nonlocal gotos. For example,

```
...
if {...} { break }
if {...} { break }
...
```

will emit two case statements, one nested within the other. The code generated for these cases quickly grows disgusting. The continuation solution, though slower, is simpler.

3.3 Balancing Parentheses

We stated earlier that Scheme's proliferation of parentheses and otherwise general lack of readability was not a big problem with machine-generated code: who wants to look at it anyway? Well, the truth is that it is an annoyance when you are trying to debug your VHLL-to-Scheme code generator. When working on Rush, we'd see the following Scheme emitted:

```
...))))))))))))
```

When these parens didn't balance with previous open parens, the Scheme interpreter would issue an error like "invalid number of arguments to *foo*", "unbalanced parentheses", "unexpected end of input" or "argument #x is not a *type*". Trying to track this from the above output can be difficult; it requires more sophisticated debugging and output. The solution is to make sure that the translator emits properly indented and commented Scheme code so that you can read it manually when debugging.

3.4 Garbage Collection and Callouts

High-quality garbage collectors interact with low-level data structures in the target language. For example, collectors generally need to know the internal data structure of each value in memory, in order to follow any pointers contained inside the structure. Providing a description for each structure is "a small matter of code", again detracting from time spent designing the VHLL. By comparison, any data structures built on top of Scheme's base structures are automatically collected properly and already have proper type tags.

Perhaps even better is the Boehm-Weiser collector, which can garbage collect arbitrary C programs with reasonable accuracy against memory leaks, and near-complete safety against corruption. The problem is that the Boehm-Weiser collector needs to scan through the entire heap to collect, because it attempts to treat *every* word in memory as a pointer to be followed, comparing that pointer against an internal table of valid data pointers. Leakage occurs when an non-pointer word accidentally corresponds to a valid heap object address, or when pointers are hidden (eg. the classic trick of XORing the "next" and "previous" pointers in a doubly-linked list to save one word per link). A worse problem than leakage is the BW collector's performance. It must run atomically; this can cause "hiccups" when used on a large heap. Boehm and Weiser suggest some ways around this, but the common solution of using an incremental collector is difficult to implement in this environment, since it requires significant cooperation from user code [BW88]. One can use virtual memory support instead, but this tends to be slow and less portable.

3.5 Lack of Sophisticated Optimizers

Machine-generated code that is not optimized typically contains many redundancies, common subexpressions, dead code, unnecessary typechecks, and so on. Ideally, to save effort, one would like to emit dumb Scheme code and have the Scheme backend perform most of the omptimizations. To do otherwise can be a hassle since complex optimizations are notoriously difficult to implement and test.

While the Scheme literature discusses many algorithms for optimizing Scheme code, and proofs have shown ways to apply many standard optimization algorithms to Scheme code [Shiv88], such facilities are far from common in Scheme implementations. Indeed, many Scheme implementations perform no optimizations at all, implying that the Scheme code presented to the compiler is literally what is output from that compiler, translated into machine code. The one exception is tail recursion elimination, mandated by R4RS, but this can be minor compared to other optimizations.

Using Scheme→C as an example target, we describe the problem faced by Rush in the above break-return-continue problem. In this case, the only way to avoid wrapping *every* statement with the trap for nonlocal gotos is to perform static analysis over the code to see what subcomputations can effect such a goto. While not difficult for the seasoned compiler optimization expert, this is rather painful in light of what should be a far easier task. In the above case, the flow equations are:

```
BREAKS(stmt)- does stmt break?

BREAKS(stmt) = case stmtType of
 IF:                        BREAKS(true-branch) v BREAKS(false-branch)
 FOR, WHILE, FOREACH:       FALSE
 BREAK:                     TRUE
 FUNCALL:                   BREAKS(arg1) v ... v BREAKS(argn)
 CONTINUE, RETURN, ... :    FALSE
```

with similar equations for each of "continue" and "return". Then, we only emit the trap if a statement could break, return, or continue. In the case of the v.1.0 of the Rush→Scheme compiler, we chose to emit a single style of case trap, so we combined the equations to check for `NONLOCAL_GOTO(stmt)`. The tradeoff is that for some cases, we produce nonoptimal case statements, as in the above example, where `'return` and `'continue` are not possible, and so can be removed from the possible goto results:

```
; a more efficient implementation of the previous example.
(case (if (> i 90) 'break)
 ((break) 'break)
 ((else (<puts "hi", increment i, recursively call loop>)))
)
```

We wish we were able to count on the Scheme compiler to detect dead code in a case statement (code bound to values that can never be produced by the predicate). Even better would be a more heroic optimizer that would reduce the above code into a single `if` statement:

```
(if (> i 90)  'break  (<puts "hi", incr i, recursive call>))
```

which may sound incredible but is well within the capabilities of better C compilers; both gcc v.2.5.8 and DEC native cc under OSF/1 v.2.1 were able to emit this reduced version for an equivalent piece of C code.

In general, the better a job the target language optimizer does, the less work you have to put into your translator to get good performance. This is as important a consideration as the final performance of your VHLL; if performance doesn't matter, neither does optimization. In the case of Rush, we claimed that performance counts inasmuch as scalability matters, but not so much that we want to try to match C. It's fine to not lose sleep about the efficiency of these case statements, but it's a bad idea to wrap every statement with such checks.

3.6 The Effectiveness of Optimization in Scheme

It is generally futile to rely on the C compiler to optimize the output of a Scheme→C translation. This is because unoptimized translator output will necessarily be tainted with false references to environments. In the loop example, our Scheme code might look like:

```
(define (foo)
 (let ((i 0))
   (letrec ((forloop (lambda ()
     (case (if (> i 90) 'break)
       ((break) 'break)
       (else (display "hi") (set! i (+ i 1)) (forloop))))))
   (forloop)))
)
```

Without analysis to determine that "i" doesn't escape the current context and isn't used later in its defining context, the Scheme implementation cannot convert this into the more efficient form:

```
(define (foo)
 (letrec ((forloop (lambda (i)
    (case (if (> i 90) 'break)
      ((break) 'break)
      (else (display "hi") (forloop (+ i 1)))))))
   (forloop 0))
)
```

In this version, "i" need not be stored in memory; it can reside in a register, leading to a performance improvement.

To test for this behavior, we passed the previous scheme code through Scheme→C and discovered that the output was more pathological than we'd thought:

```
TSCP t_foo()
{
  TSCP X2, X1;
  X1 = _TSCP(0);
  X1 = CONS(X1, EMPTYLIST);
L2040:
  if (LTE (_S2CINT (PAIR_CAR (X1)), _S2CINT (_TSCP (360)))) goto L2044;
  X2 = x2013;
  goto L2045;
L2044:
  X2 = FALSEVALUE;
L2045:
  if (EQ (_S2CINT(X2), _S2CINT(c2013))) goto L2042;
  scrt6_display(c2015, EMPTYLIST);
  X2 = _TSCP (PLUS ( _S2CINT (PAIR_CAR (X1)), _S2CINT ( _TSCP (4))));
  SETGEN (PAIR_CAR (X1), X2);
  GOBACK (L2040);
L2042:
  return (c2013);
}
```

In this code, X1 ("i" in the source) is represented by a pair, where a direct reference would do; the C compiler is unable to detect and eliminate this unnecessary level of indirection.

How does this affect the VHLL author? Consider that the above loop, when we exclude the display call, essentially counts from 0 to 90. The "fast" version took 85μsec on a 486dx2/50 running Linux; the slower version took 145μsec. On the Alpha, the times were 22μsec and 41μsec. Considering the number of such "minor" optimizations that apply to any given situation, the VHLL implementor cannot ignore them if he cares about the speed of his language. It is not sufficient to rely on the machine code translation process to decrease runtimes.

4 Rush as an Example Language

The Mariposa database project needed an extension language with the feel of Tcl but which also ran quickly (within a factor of 100 of the speed of C). Faced with a time table allowing us five months to go from design to demo, we were left with two options: either finish the existing Tcl Compiler

(TC) and add the features we needed, or build a new language from scratch. TC was too slow for our needs; trying to use it would have been far worse than embarking on a new project.

So far, the Rush project has been very successful. Not only did we produce a working interpreter and optimizing compiler, but at our release demo we operated "hands free", letting users play with the system directly. What makes this exciting to us is that at no time have we had to concern ourselves with the way procedures are declared, invoked, or stored. We've never had to worry about the semantics of our memory manager or debugging its implementation; Rush inherited a generational copying collector from our chosen Scheme implementation. We have never seen the assembly output from the Rush compiler; we never needed to.

Here we will describe some of what we did with Rush. This section is divided into two parts. The first part talks about Rush rules (the most Very High Level feature of Rush) and "boxing", the technique we used for implementing them. The second part discusses optimizing away the need for these boxes to increase language performance.

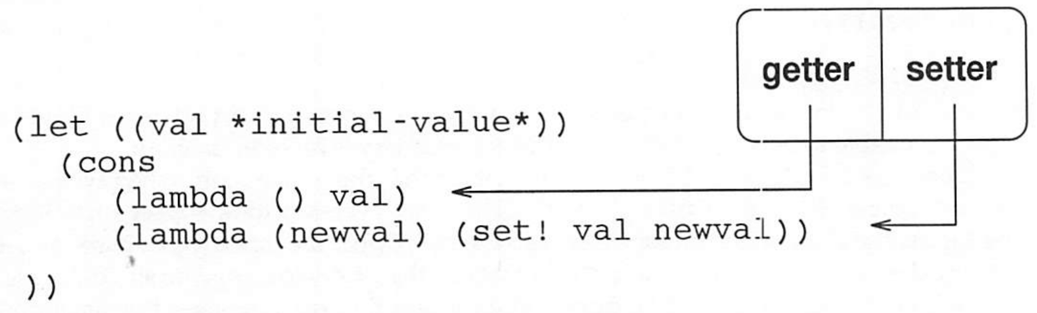
4.1 Boxing

To provide a more in-depth case study of what we found useful in Scheme, we present the design and implementation of "boxing" and how it works in Rush. From the beginning Rush was required to support commands of the form "on <event> do <action>", where <event> is an arbitrary predicate and <action> is an arbitrary block of statements. These "production rules" are sort of like "if" statements that are always watching. Here is a sample rule:

```
    on {x>3 && y<10} do {notify x y z}
```

The action `[notify x y z]` will be performed whenever the predicate `[x>3 && y<10]` becomes true *any time over the course of program execution.* For a more in-depth description of how these rules work and why they're useful, see [SBD94]. The most intuitive way to implement rules is to monitor the changes in relevant variables. For this we chose a *boxing* mechanism.

Rush "boxes" are pairs of functions, a "getter" and a "setter", which are called to retrieve values from and store values to a memory location. Rush semantics require that every variable behave like a box. For normal variables, the getter and setter access the location directly, like one would expect. Figure 1 shows a diagram of a getter/setter pair for a boxed variable "foo" that is doing nothing tricky.

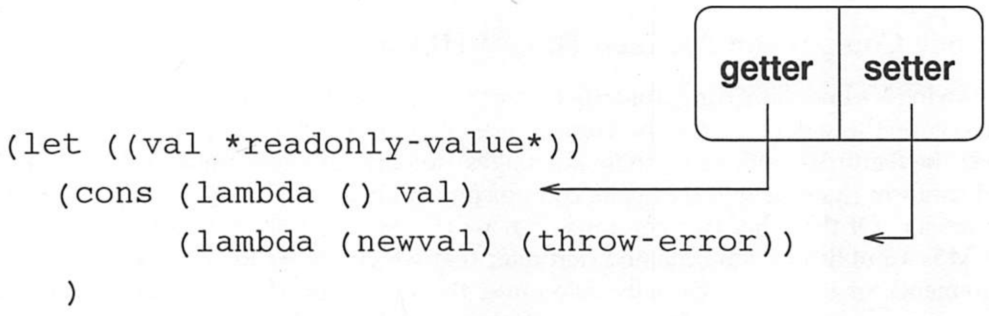


```
(let ((val *initial-value*))
   (cons
      (lambda () val)
      (lambda (newval) (set! val newval))
 ))
```

Figure 1: A Rush variable (the box is a Scheme "pair" of pointers.)

If we wish to create a read-only variable, we just modify the setter of its box so that instead of changing the value, the setter generates an error. This configuration is shown in figure 2. Likewise we can easily imagine a "read trap" as a modified getter function.

These boxes are easy to create and manipulate because of Scheme's first-class functions and automatic memory management.



```
(let ((val *readonly-value*))
   (cons (lambda () val)
         (lambda (newval) (throw-error))
   )
```

Figure 2: A read-only variable in Rush. The overall function still returns a box. When its getter is called, it errors; when its setter is called, the box is rewritten to be a normal variable.

We also use boxes to implement rules. A rule is represented by a set of functions which are added to the setters of relevant boxes. Since the rules are hidden inside boxes, they are never seen unless the boxes they effect are modified. Therefore the implementation scales well; that is, programs do not slow down as rules are added.

One beautiful feature of rules that proceeded naturally from their implementation in Scheme is that they are scoped; a rule has the same scope as the box it is attached to. Here is some sample Rush code that uses a local rule:

```
proc foo (list m) {
    on {[llength m] == 0} do {error "list has run out of elements!"}

    i = 0
    while {i < 100} {
        if {member? i m} {remove i m}
        i++
    }
}
```

If the list m ever becomes empty while evaluating foo, an error will be thrown; but if foo exits without throwing an error, the rule will be forgotten because it is out of scope.

4.2 Optimizing Away Boxes

If every variable in Rush were actually boxed, at least one Scheme procedure call would be necessary to get the read a variable's value; this is an unacceptable performance penalty. Therefore we adopted a convention that variables may be either boxed or unboxed. We created a macro called `get-value` which checks to see whether a variable is boxed; if so, the getter is called, otherwise the value itself is used. Then we modified the code generator to make sure that only values that needed boxes (that is, values whose getters or setters did something unusual) were ever boxed. Since `get-value` is a simple `if` statement, it compiles to fast code in the general case (the case where a value does not need to be boxed).

5 VHLL Comparisons

First we present a breakdown of the effort and line-count of the Rush source code by topic of implementation, comparing it to the source trees for a few other public domain VHLLs. Then we show a few performance figures for Rush in an attempt to demonstrate that our use of Scheme as an IR actually helped us make a *faster* language than coding directly in C. We've discussed many of Scheme's virtues and shortcomings; now we present empirical results.

5.1 Code Comparison Between Three VHLLs

The following is a line count comparison between the source trees of a set of VHLL implementations. In some sense, this comparison is inaccurate since in almost every case we're comparing apples to oranges: the features provided by these languages are fairly different from each other. Both Rush and Tcl conform rigorously to the Sprite coding conventions[Ous89], which include a fair volume of comments. Of the other two columns, Perl v4 is a byte-compiling interpreter by Larry Wall and SCM is a public domain Scheme interpreter that we've ported Rush to. Please note that the measurements for Perl4 are especially inaccurate; the source code has little documentation so we have based our figures on educated guesses. We have inflated the line counts for Perl to compensate for the lack of comments.

You may notice that the Rush parser is large. This reflects our efforts to unify the command syntax of Tcl with the operator syntax of C, so as to minimize the verbosity of the language. Details will be provided in a forthcoming technical report. Again, the point is that our use of Scheme freed us to explore this aspect of the language that we would otherwise not have had time for.

VHLL implementation sizes (in lines of code)
n/a = feature not provided by the VHLL.

description	perl4	Tcl7.3	Rush	SCM4d3
read-eval-print loop	4,700	2,300	100	350
main and runtime lib	2,500	3,600	1,500	3,200
parser	5,000	1,300	5,600	200
memory management	600+	600	n/a	1,900
regexp library	3,300	1,900	Tcl	250
arithmetic library	300	2,300	300	1,900
basic data structs, types	4,600	1,800	2,200	4,800
compiler/codegen	3,000	n/a	2,500	n/a
strings and conversion	3,700	2,600	450	n/a
var traces/rules	n/a	1,700	1,500	n/a
exception handling	300+	300+	660	300
total of all rows	28,000	18,400	14,800	12,900
total for language	40,700	26,700	17,600	20,000

Some notes:

- Rush entries labeled "Tcl" are stolen from an embedded Tcl interpreter, which is linked to Rush as a shared library and called through Tcl's C interface.
- The "memory management" row only accounts for the central controller of memory. Perl and Tcl don't seem to offer fully-automatic memory management, so some memory management code exists outside the core. The large size of Scheme's memory management facility is due to the sophisticated garbage collection algorithm.
- "Basic data stucts and types" include hash tables, lists, booleans, and common operations on them. This mostly demonstrates that while Scheme can save some of the labor, a typical VHLL still needs to tailor primitive types to its purposes. In the case of Rush, boxing required us to duplicate much of this sort of code.

- In most systems, error handling code is distributed across many places in many functions; these estimates are surely undercounts of this difficult-to-measure item. Again, the point is that a new VHLL will likely need its own exception and error handling mechanisms.
- Again, these numbers are *very* rough estimates, and shouldn't be used except to say that a feature took "a lot" or "a little" code to implement in the given language. Assume that they may be off by 25-50%. Each system has 6,000-10,000 lines missing; they are typically from "portability and OS interface", and reflect more how widely ported the system is (or how portable the code is) and what OS interface features are implemented (ie. networking).

5.2 An Example: Conway's Life

We implemented Conway's game "Life" using the naive $\theta(n^2)$ algorithm commonly used by novice programmers and by experts prototyping such an program. Some notes:

- "tcl" refers to the Tcl7.3 implementation. "rush-interp" refers to a nearly-identical version for Rush (syntactic changes only). "rush-compiled" refers to the same code compiled into Scheme, then into C by Scheme→C.
- "rush-compiled" is still a Rush interpreter, but with the time-consuming set of Life primitives compiled into the executable. This is directly analogous to Tcl, which lets users bind Tcl commands to C functions; in Rush, we also let users compile Rush functions into the interpreter. Like Tcl, they can be redefined at runtime.
- We consider these numbers very poor for Rush, owing to the high number of assignments done in the program code. Such assignments (eg. to update the next generation's board) incur high overhead because they interact unpleasantly with Scheme→C's garbage collector.

These comparisons are highly Tcl-centric because we developed Rush as a descendant of Tcl. Additional benchmarks can be found in [SBD94]. While we didn't measure other systems such as Perl4 or Python, various accounts from the comp.lang.tcl USENET newsgroup suggest that their speeds are similar to Tcl's.

10x10 board size			20x20 board size		
tcl	1560μsec	1.0x Tcl	tcl	8400μsec	1.0x Tcl
rush-interp	477μsec	3.3x Tcl	rush-interp	1880μsec	4.4x Tcl
rush-compiled	62.2μsec	25.1x Tcl	rush-compiled	264μsec	31.0x Tcl

40x40 board size		
tcl	65200μsec	1.0x Tcl
rush-interp	7600μsec	8.5x Tcl
rush	1060μsec	61.0x Tcl

5.3 Other Languages Besides Scheme?

As time goes on we see that other languages might make IRs as effective as Scheme. Probably the most promising of these is ML.

We had several reasons for originally choosing Scheme over ML. Scheme has a simpler syntax which, as we pointed out, makes code generation easier; Scheme's stated claim of orthogonal functionality seemed like a pretty good idea for a high-level IR.

ML compiles to very efficient code, usually more efficient than Scheme. However, much of this efficiency is due to ML's system of strong typing— that is, that the type of an identifier in ML must be known at compile-time and must remain constant. For Rush, we wanted to offer dynamic typing: we wanted identifiers that could always hold values of any type, and that could change type on the fly. This means that in the ML output code of our compiler each identifier's type would have to be the union of all types. This would keep the ML type inference system from being able to infer anything useful, thus discarding much of the point of using ML.

There were also far more Scheme implementations to choose from than ML. When we started this project, the only full-scale optimizing ML compiler we knew of was Standard ML of New Jersey (SML/NJ). Whereas SML/NJ is a high quality implementation, it is also quite big. At the same time there seemed to be many viable Scheme implementations, most of them fairly small. Since then, two smaller ML implementations have been announced for a variety of platforms, CAML and Moscow ML.

As time goes on, we expect ML to be a much more viable alternative to Scheme; we also expect other languages to become good possibilities.

6 Conclusions

Scheme offers the VHLL designer many advantages which together can drastically reduce the effort needed to design new languages. The same arguments can be used to claim that Scheme provides similar benefits to the modifier of a VHLL, since more of the underlying engine is "standard" (non-proprietary).

There are disadvantages to using Scheme, but the benefits seem to outweigh them. With Rush we found the savings dramatic, compressing a multi-year development cycle into a relaxed half-year; this included a parser, an interpreter, a compiler, a binding to the Tk graphical user interface library and a binding to the Tcl-DP distributed computing library, and ports to several platforms and two Scheme back-ends.

Much of this paper focused on performance issues. Our stance concerning language performance is fairly simple: if you're going to make changes to the design or implementation that cause the resulting system to be slower, it should be for a good reason. Adding "high level" features, making the system easier to use, etc. are all reasonable excuses to slow the system down. In the implementation of VHLLs, performance has often been traded away for greater ease of design and speedier implementation of the VHLL. We suggest that Scheme provides the VHLL designer with a feature-rich, fast environment, obviating the need for this tradeoff.

6.1 Future Work

In the future we're looking to expand and complete the Rush environment. Part of this will necessitate building a debugger, another painfully laborious project that we'd like to wish into existence (as opposed to "debug into existence"). We have no idea how we're going to leverage off of existing debuggers, since they tend to be language specific, but that's never stopped us before, and the deadline is even worse this time.

Students and researchers interested in the system should send email to the authors. We do not expect to publicly release the system until we're satisfied that its syntax and semantics will remain fairly constant.

7 Biography

Adam Sah is a PhD student at UC Berkeley, working on the Mariposa massively distributed database. Rush will serve as the "glue" language and also as the expert system shell in which we will encode distributed storage management policies. His other interests include poetry, raving, and lazy afternoons with his cats.

Jon Blow is an about-to-be-paroled undergrad at UC Berkeley who's worked on an update to INTERCAL, the FMPL language, and most recently Rush. He's never commented code with `/* more deep magic */` or `/* you are not expected to understand this */`

8 Acknowledgements

The authors graciously acknowledge the work of Brian Dennis, who helped on the Rush prototype, and to Raph Levien, who suggested the use of the getter-setter mechanism. Thanks also to the program committee, whose feedback was invaluable, to John Ousterhout, Mike Stonebraker, Sue Graham and Paul Hilfinger, who each guided our understanding of the issues involved in the Rush language design.

References

[App92] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. "Compilers: Principles, Techniques and Tools." pp.422-423 introduce the issues in implementing dynamic scoping, although their text predates the acceptance of RISC architectures and the new importance on storing local variables in registers, as opposed to their maintenance in memory locations.

[Bart89] Joel Bartlett. "Scheme→C: a portable Scheme-to-C compiler". DEC WRL Technical Report #89/1, Jan, 1989.

[BT93] Bill Triggs. AVCALL 0.1 Documentation. ftp.robots.ox.ac.uk: /pub/gnu/avcall0.1.tar.gz

[BW88] Hans-Juergen Boehm and Mark Weiser. "Garbage Collection in an Uncooperative Environment." Software- Practice and Experience, Vol 18(9), Sept. 1988.

[CFRWZ91] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." ACM Trans. on Prog. Lang. and Systems, 13:4, October, 1991.

[DFH86] R. Kent Dybvig, Daniel P. Friedman and Christopher T. Haynes. "Expansion-Passing style: Beyond Conventional Macros". ACM Conf. on Lisp and Functional Prog. 1986.

[Ous89] John Ousterhout. "The Sprite Engineering Manual." UC Berkeley Technical Report #89/512, 1989.

[R4RS] William Clinger and Jonathan Rees, ed. "The Revisedâ Report on the Algorithmic Language Scheme." Lisp Pointers IV (July-Sept 1991).

[RZ88] Barry Rosen and Kenneth Zadeck. "Global Value Numbers and Redundant Computations." ACM Princ. of Prog. Languages, January, 1988.

[Sah94] Adam Sah. "An Efficient Implementation of the Tcl Language". Master's Thesis, Univ. of Cal. at Berkeley tech report #UCB-CSD-94-812. May, 1994.

[SBD94] Adam Sah, Jon Blow and Brian Dennis. "An Introduction to the Rush Language." Proc. Tcl'94. June, 1994.

[SF89] George Springer and Daniel P. Friedman. *Scheme and the Art of Computer Programming* MIT Press, Cambridge, MA 1989.

[SG90] James W. Stamos and David K. Gifford. "Remote Evaluation." ACM Trans. on Prog. Lang. and Systems, Vol 12, No. 4, October, 1990.

[SHH86] Michael Stonebraker, Eric Hanson, and Chin-Heng Hong. "The Design of the Postgres Rules System." UC Berkeley tech report #UCB/ERL M86/80.

[Shiv88] Olin Shivers. "Control Flow Analysis in Scheme". ACM Prg Lang Des. and Impl. June, 1988.

[SICP] Harold Abelson, Gerald Jay Sussman and Julie Sussman. *Structure and Interpretation of Computer Programs* MIT Press, Cambridge, MA 1985.

[Ston93] Michael Stonebraker. "The Integration of Rule Systems and Databases." UC Berkeley tech report #UCB/ERL M93/25. 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. "Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions." Conf. Proc. of ACM Princ. of Prog. Lang., Portland OR, 1994.

[VP89] Steven Vegdahl, Uwe Pleban. "The Runtime Environment for Screme, a Scheme Implementation on the 88000". 3rd Int'l Conf. ASPLOS. SIGPLAN Notices 24, April 1989.

[Wil92] Paul R. Wilson. "Uniprocessor Garbage Collection Techniques." Intl. Workshop on Memory Management. St. Milo, FR. Sept, 1992.

[WL73] Donald Woods and James Lyon. "The INTERCAL Programming Language Reference Manual." Technical report. Stanford University, 1973.

NOTE: There are no pages 39-40. The next page is page 41.

Tcl/Tk for a Personal Digital Assistant

Karin Petersen
Computer Science Laboratory
Xerox PARC

petersen@parc.xerox.com

Abstract

This paper reports on the experience of providing Tcl/Tk for the ParcTab, a personal digital assistant built at Xerox PARC. The primary reason for supporting an extension language like Tcl/Tk for a PDA is to supply the same platform-independent infrastructure for user interface design and communication between the PDA and remote applications. The result was that Tcl/Tk enabled rapid prototyping and customization of applications for the Tab, most of which were extensions and interfaces to existing non-Tab applications. In addition, by using a platform-independent extension language, interfaces designed in Tcl/Tk for the Tab are able to be reused on other platforms that provide a Tcl/Tk implementation.

The paper starts with a discussion of the decisions made during the process of porting Tk to the ParcTab, which were focused on maintaining the natural look and feel of the Tk widgets while exploiting the small area of the display as much as possible; then includes a description of some applications that were enabled by Tcl/Tk on the ParcTab; and finally, presents a summary of some tradeoffs available to the application designer for personal digital assistants with small displays.

Keywords: extension language, personal digital assistant, distributed application, user interface, application generation.

Introduction

In the emerging world of ubiquitous computing, many computers of different sizes coexist at any moment in a user's environment. At Xerox PARC, we have built such devices, ranging from the Liveboard [EBG+92], an electronic whiteboard with a 46" ×32" screen, to the ParcTab, a hand-held personal digital assistant with a 2.5" × 2" display [SAG+93].

For ubiquitous computing [Wei91] to become a reality, these different kinds of computers must interact with one another and with other devices in the environment. For example, the Liveboard can be used locally to collaborate with other people working remotely from their workstations; the ParcTab can advance slides presented on the Liveboard with an on-line presentation manager; the ParcTab can control video equipment in a meeting room. These examples illustrate that the strength of the multiplicity of computing devices resides in their capacity to be used together to perform a task.

Software application and system designers are also increasingly aware of the need for supporting distributed control of applications. Such support exists either on a per application basis, or through particular platforms or protocols, such as VisualBasic, AppleEvents, and OLE, as well as global standards, such as CORBA [Ude94]. The trend is towards opening up applications in order to control and extend one application with others.

Personal Digital Assistants (PDAs), like the ParcTab, can provide particularly interesting added value to applications that are open to remote interactions. Interfaces for these applications can be designed for the PDA, and a user may then interact with the application through these interfaces.

The Liveboard presentation management from the PARCTAB is an example of such an interaction between a PDA-based interface and an application on another computer. Note that in this particular example the added value is crucial, since it has been found that widget type interfaces for the Liveboard are awkward to use [PL92].

The design of an infrastructure that supports remote interfaces to applications from a PDA can be separated into two parts: a mechanism to generate the graphic user interfaces (GUIs) for the PDA, and a communication mechanism between the PDA and the applications with which it is interacting. General Magic has taken this approach in their design of Magic Cap as a user interface abstraction and Telescript for communication [GM994]. At Xerox PARC we are also exploring this design space. We use an extension language and its windowing toolkit, Tcl and Tk [Ous94], as the common infrastructure for PDA user interface programming and as the communication platform between the PDA and remote applications.

This paper reports on the experience of providing Tcl/Tk for the PARCTAB. Basically, the prototyping flexibility of Tcl/Tk enabled a number of applications for the PARCTAB in a few months, primarily because extensions and interfaces to existing non-Tab applications became simple. The first few sections of the paper focus on the design decisions made when porting the Tk windowing toolkit to the 2.5" × 2" display of the PARCTAB, including decisions regarding issues of uniformity between the look and feel of widgets on workstation displays when compared to the widgets on the PDA. A number of the widgets implemented for the Tab are discussed in light of these issues.

The paper then points out the advantages of using Tcl/Tk rather than a platform-specific scripting language. I present the results of experimenting with the reusability of user interfaces designed for the Tab on a workstation implementation of Tcl/Tk. Finally, the focus of the paper turns to the issues regarding the creation, ease of prototyping and customization of user interfaces and applications for a PDA like the PARCTAB.

Why add Tcl/Tk to a PDA?

In the last couple of years many different types of small hand-held computers have emerged in the market. They vary mostly in size, style of input, communication capabilities, and supported applications. Although the sizes of the PDAs varies, they normally stay in the range of 5" × 7". The size of the display depends on the type of input mechanism that the device provides. Stylus-based PDAs, such as the Apple Newton, can afford to have larger displays than keyboard-based PDAs, like the Casio Boss, for roughly the same size device. The type of applications available on PDAs varies, with most providing calendar, address book, notebook and calculator operations. In addition, some systems include personal finance applications, like Quicken, while others come with the DOS operating system, which provides a basis for many other applications.

In general, the communication capabilities of the PDAs vary significantly. Many models can be tethered to PCs or Macintoshes in order to download and upload the databases maintained on the PDA. More recently wireless communication mechanisms are also included in some systems. The Apple Newton provides an infrared beaming mechanism, and PCMCIA cards with pagers are also available for the Newton. RadioMail Co. connects cellular phones and software to HP-95 systems, providing support for instant message delivery to these devices. The PARCTAB has diffuse infrared networking capabilities.

Most PDA systems provide different functionalities when connected to other systems. As mentioned above, when in connected mode the value of PDAs comes from being able to interoperate with other applications or the environment. Currently the most common functionality provided by PDAs in connected mode is to transfer data from applications like calendar managers and address books between the user's stationary and mobile computing platforms. Even some wristwatches provide this functionality today [Kim94]. This data transfer operation is useful to support consistent views of the user's data across the different platforms.

However, the degree to which PDAs will be used depends on how seamlessly and extensively they can interoperate with many other applications. With the imminent explosion of wireless communica-

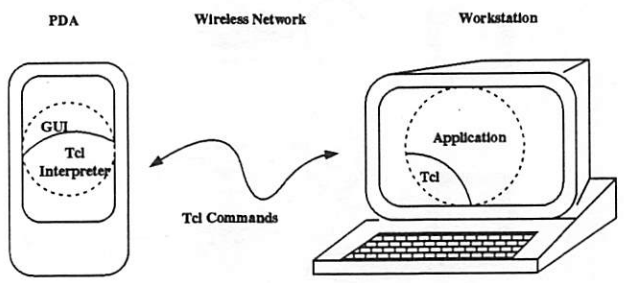


Figure 1: Tcl as the link between PDA GUIs and applications.

tion technology. PDAs could be used to access information produced by establishments like shopping centers, retail stores, and museums; to control applications like presentation managers, video and audio devices, telephone systems, and, of course, many others. Since interoperability between the PDA software and remote applications is the key for the success of this type of use of hand-held devices, the infrastructure for interoperability is very important.

On the other hand, more and more applications are exporting a subset of their functionality, either explicitly through shell commands, extension languages like Tcl and Icon, or through protocols like OLE and AppleEvents. An extension language, such as Tcl, can be used to link the PDA software with other applications. Tcl, complemented with its windowing toolkit Tk, provides a high level language which can be used to program the graphical user interfaces for the PDA. In addition, because Tcl can be embedded directly into applications, it also provides a high level communication platform that allows commands to be sent to Tcl interpreters of other applications. Figure 1 shows the architecture of a PDA system interacting with another application extended with a Tcl interpreter to handle commands issued remotely.

The advantages of using an extension language and its windowing toolkit, like Tcl/Tk, as the infrastructure supporting interoperability between PDAs and remote applications are:

- The scripting language is an effective mechanism for fast application prototyping;
- Tcl/Tk is implemented on several different computing platforms, therefore the opportunity exists to reuse interfaces designed on different platforms;
- only *one* language is required to implement the user interface and communication aspects of applications, both on the PDA side and on the application side;
- finally, end users can extend and customize any part of the system that they control, without having to rely on drag and drop mechanisms which are nearly impossible to implement for small screen devices.

Currently, some PDAs provide a scripting language for user interface design and communication. For example, NewtonScript [Smi94] is part of the application development kit shipped with the Apple Newton. On the other hand, scripting languages like Telescript have been designed solely to handle communication to and from the PDAs. However, using one language that is available on multiple platforms to combine the two has not been explored.

Tk for the PARCTAB

This section describes the PARCTAB system architecture and the Tk implementation for it. Basically the approach taken to port Tk to the PARCTAB, also just called the Tab, was to borrow the Tk

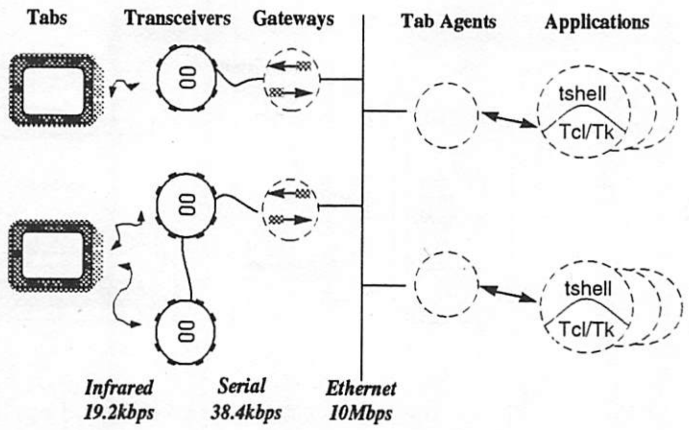


Figure 2: The PARCTAB system (dashed lines indicate software processes).

syntax for interface specifications and reimplement each of the commands for the Tab. The main challenge of the implementation was to manage the Tab's small display effectively.

Most of the decisions made during the Tab-Tk design process focused on maintaining the natural look and feel of the Tk widgets, while exploiting the small area of the display as much as possible. The following were the key observations and decisions made during that process:

- The PARCTAB screen is too small to display multiple windows at the same time. The screen management is therefore centered around a "one window at a time" philosophy.
- Menus provide a natural mechanism to design user interfaces that organize the presentation of choices to the end user. Because of the limited screen area on the Tab, menus will be frequently used to structure user interfaces and therefore need to be intuitive to use and have good response times.
- The PARCTAB size and processing capabilities call for simplicity in the design. The current implementation of the Tk toolkit for the Tab uses the placer for geometry management and provides buttons, labels, menus, text, entries, frames and toplevels as the core widget set.

The PARCTAB System

The PARCTAB is a stylus-based PDA. It has a 2.5" × 2" touch-sensitive display and three vertically arranged external buttons, TOP, MIDDLE, and BOTTOM. The Tab communicates with stationary infrared (IR) transceivers via diffuse infrared packets. The IR packets travel between the Tab and the IR transceivers, which are in turn connected to workstations through a serial link. The infrared network is designed for in-building use. In most cases, there is one transceiver in each room, making a room the basic communication cell. Figure 2 shows the hardware and software components of the PARCTAB system.

There are three types of software components in the system: gateways, agents and applications. The *gateways* run on the workstations that connect to an IR transceiver with a serial link; a gateway processes packets received by the IR transceiver and forwards them to the appropriate *agent*, conversely the transceiver receives packets from agents destined for Tabs and forwards them to a room's IR transceiver. Each PARCTAB is represented by an *agent*, which tracks the location of the device and provides location-independent reliable remote procedure calls for the Tab. A distinguished application, called the `shell`, is created by the agent and allows users to start and switch between applications.

The PARCTAB system was designed to enable experimentation with continuously connected ubiquitous computing devices. At Xerox PARC, we were interested in identifying both the basic components of the infrastructure for a system with such devices and the type of applications that continuous and ubiquitous connectivity facilitates. Because the Tabs are continuously connected to the rest of the computing network, it was an early engineering decision to fabricate the Tab with a low-power processor and offload the execution of Tab applications to remote hosts. In that sense the Tab can be viewed as a mobile terminal. However, the lessons learned from designing user interfaces and communication support for Tab applications applies to other PDAs regardless of where the application code actually gets executed.

The Tab system software is implemented in Modula-3 [Nel91]. Originally, a library of widgets designed to handle the Tab's low infrared bandwidth and the small display area was available for application implementation. This library, called TabBits, was used to write Tab applications with graphical user interfaces much the same way people write applications for X11 with language specific windowing toolkits. Hence, programming Tab applications using TabBits was complicated exactly for the same reasons it is complicated and awkward to program applications for X11 at this level: the program designer has to focus on low level properties of the window system, getting too far removed from the abstract view of the application's intended functionality.

Designing user interfaces at a low level is particularly troublesome for applications that have very simple graphical interfaces that get used to interact with applications in the surrounding environment. Since this type of application was frequently designed for the Tab, the need to support the implementation of these applications at a higher level became apparent. Examples for such Tab applications are: an e-mail reader, an interface to on-line control of electrical appliances, a file system browser, a remote interface to a presentation manager, remote control of the cursor position on a large display, and others.

This observation motivated the current work: the creation of a high level platform to program Tab applications. The goal was to provide the ParcTab with a scripting language that supports remote communication and a windowing toolkit. I chose Tcl/Tk for this purpose. Although Tcl and Tk are not inseparable and, in fact, several other scripting languages, like Python and Scheme, support the Tk windowing toolkit, the choice of Tcl/Tk was mostly based on three reasons:

1. Tcl/Tk was already widely used among groups within and outside of Xerox PARC.

2. Tk provides a complete set of building blocks for creating graphical user interfaces. This enabled me to select and implement a subset of widgets useful for the ParcTab's small display size.

3. Tcl/Tk can be embedded into applications, thereby providing a high level communication platform, which allows commands to be exchanged between Tcl interpreters of several applications.

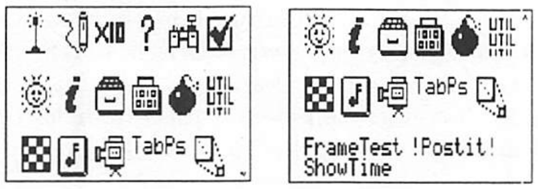
For the experiment of using Tcl/Tk for the design of graphical user interfaces and as the communication platform for the Tab, I implemented the `tshell` to replace the original shell. This replacement was very easy since the ParcTab system was designed early on to allow any application to provide the `shell`'s functionality of selecting and switching between Tab applications. The tshell consists of a Tcl interpreter, Tab-Tk, a subset of the Tk windowing toolkit, and a set of special commands to handle some of the Tab-specific events and functions. At startup time, the tshell reads in a Tcl script called `.tabrc.tcl`, which initializes the interfaces to applications on the Tab.

Tab-Tk

Tab-Tk is implemented on top of the Modula-3 TabBits widget library. The next few subsections describe some of the widgets supported by Tab-Tk, and how they compare to the original implementation of Tk for X11.

Toplevels for Screen Real-Estate Management

The ParcTab screen has 128×64 monochrome pixels, which is over 90% smaller than the display of a modern workstation. For such a small display, it is impractical to have multiple windows coexisting



```
.main add cascade -bitmap @comms.xbm -menu .main.comms
.main add cascade -bitmap @draw.xbm  -menu .main.pen
.main add cascade -bitmap @x10.xbm   -menu .main.x10
.main add cascade -bitmap @location.xbm -menu .main.location
.main add cascade -bitmap @desk.xbm  -menu .main.desk
.main add command -bitmap @$icon(tabarbitron) -command { ... }
.main add command -bitmap @sun.xbm  -command { ... }
.main add command -bitmap @info.xbm -command { raise .help }
.main add command -bitmap @tabbrowse.xbm -command { ... }
.main add command -bitmap @$icon(tabloader) -command { ... }
.main add cascade -bitmap @bomb.xbm     -menu .main.new
.main add cascade -bitmap @utils.xbm    -menu .main.utils
.main add cascade -bitmap @gamesGR2.xbm -menu .main.games
.main add cascade -bitmap @tunes3.xbm   -menu .main.games.tunes
.main add cascade -bitmap @movies.xbm   -menu .main.movies
.main add cascade -label  TabPs         -menu .applist -command listtabapps
.main add cascade -bitmap @$REMOTEWIN/openwin.xbm -menu .main.remotewin
.main add command -label  "FrameTest"  -command { ... }
.main add command -label  "!Postit!"  -command { ... }
.main add cascade -label  "ShowTime"  -menu .main.slideshow
```

Figure 3: Example of a menu.

on the screen. Therefore, one of the first decisions made when implementing Tk for the Tab was to assume a stack model of top-level windows.

In Tab-Tk, a top-level window is equivalent to the PARCTAB "screen", which is created with the **toplevel** command. The **raise** command is then used to alternate between top-level windows. Abstractly, toplevels on the Tab are stacked like a deck of cards, and one at a time can be raised to the top of the deck and displayed on the screen. The toplevel window and the raise command are therefore the basic components of the PARCTAB screen management. This approach is similar to the hypercard model.

Unlike in Tk, toplevels in Tab-Tk are always the size of the full screen. The size arguments of a toplevel are ignored when processing the command, which allows user interfaces written for the Tab to be reused on devices with other screen sizes.

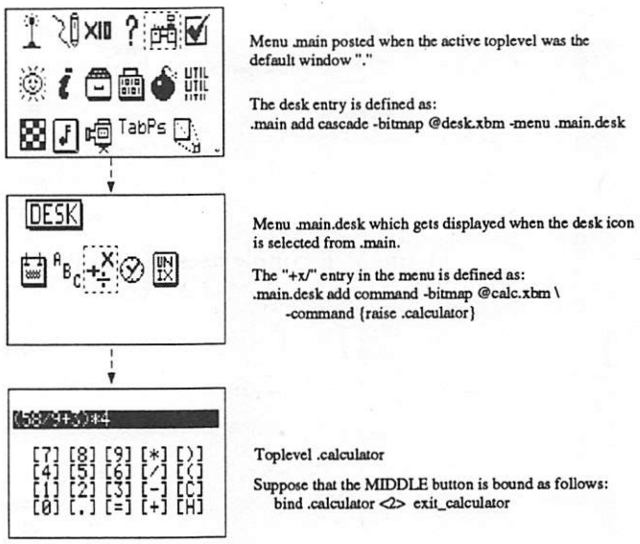
Menus

The next important widget for user interface design provided for the Tab is a menu. It provides the basic mechanism to organize choices presented to the user. A menu is equivalent to a list of text or bitmap buttons that trigger an action when selected with the stylus. The menus in Tab-Tk are different from those in Tk in that as many entries as possible are placed in a row. By doing so, the limited screen real estate of the PARCTAB can be better exploited. This placement of entries in a menu is automatic.

The TOP and BOTTOM buttons of the Tab are used to scroll over menus that have more entries than can fit on the screen. Small arrow heads in the top and bottom right corners will indicate whether the menu has entries beyond the screen. Figure 3 shows an example of a Tab-menu. The second view of the menu results from scrolling towards the last three entries not visible in the first view, as indicated by the arrows.

Similarly to toplevel windows, a posted menu occupies the whole PARCTAB screen. The MIDDLE button is used to exit from a menu, this causes the menu to be unposted and redisplays the screen that was active before the menu was posted.

Selecting a menu-entry with the STYLUS will cause the command associated with the menu entry to be invoked. In addition, if the selected menu-entry corresponds to a submenu entry, that is, has the "cascade" type, the menu specified in the entry definition will be posted. Navigation trough the



Menu .main posted when the active toplevel was the default window ".”

The desk entry is defined as:

```
.main add cascade -bitmap @desk.xbm -menu .main.desk
```

Menu .main.desk which gets displayed when the desk icon is selected from .main.

The "+x/" entry in the menu is defined as:

```
.main.desk add command -bitmap @calc.xbm \
      -command {raise .calculator}
```

Toplevel .calculator

Suppose that the MIDDLE button is bound as follows:

```
  bind .calculator <2> exit_calculator
```

Figure 4: Example path between the use of cascading menus and toplevels.

menu hierarchy is therefore based on selecting a new menu by pointing at the appropriate menu-entry with the STYLUS, and unposting the menu by clicking the MIDDLE button.

Screen management - one step further

Since both menus and toplevels occupy the entire screen, when they are displayed, it is necessary to provide an efficient and intuitive mechanism to indicate what should be displayed on the PARCTAB screen when a toplevel or a menu gets raised, posted or unposted. For cascading menus the problem is fairly simple, when a submenu gets unposted the window (menu or toplevel) from which the menu was posted gets redisplayed. However, there is no such explicit link between the raisings of several toplevel windows, nor between menus and toplevels.

Consider the execution illustrated in Figure 4. In the example, the MIDDLE button of the toplevel window `.calculator` is bound to a command used to exit the calculator implementation. In other words, when `.calculator` is in the foreground of the PARCTAB screen and the MIDDLE button is pressed the `exit_calculator` command should cause a different window to be raised. The question is which one? It could be the window that was active when `.calculator` was raised, that is, in this case it would be sufficient that `exit_calculator` execute `raise .main.desk`.

However, if the calculator were to be invoked from different windows or menus, this solution would be very static and unsatisfactory. I evaluated several alternatives to specifying what window should be raised when the implementor wants to preserve the ordering by which windows are invoked:

1. Use raise with the explicit window name, as described above;
2. Using a global variable named `prevWin` set by the Tab-Tk runtime;
3. Add a `leave` command for toplevels, which redisplays the window that was active at the time the last raise command was executed for the toplevel.

The `prevWin` global variable is set to the currently active window name when a raise or menu post command gets executed. The execution of `raise $prevWin` will therefore redisplay the toplevel window or menu that was in the foreground when the previous raise was executed. The use of the `prevWin` variable overcomes the limitation of statically specifying to which window to return from a toplevel. However, if a sequence of multiple toplevels gets raised, the variable gets overwritten and therefore, to preserve the display hierarchy of the windows, `prevWin` needs to be explicitly saved

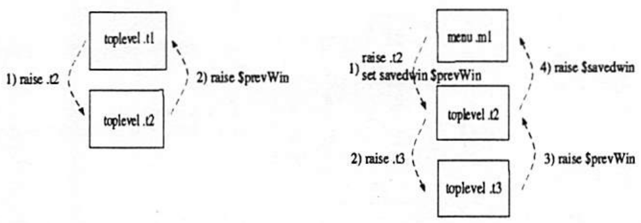


Figure 5: Example uses of prevWin.

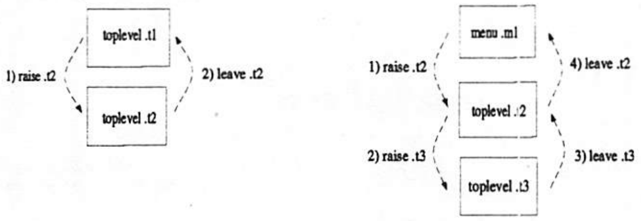


Figure 6: Using the raise and leave commands.

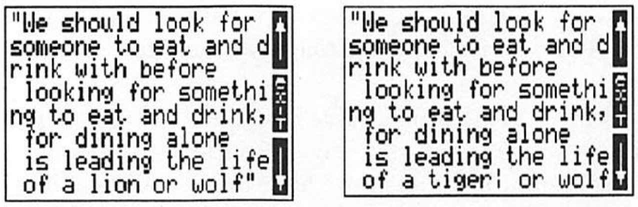
after an execution of the raise command (see Figure 5). This constraint can easily lead to subtle bugs in the user interface implementation, either because the variable is not saved or because it is saved in a procedure that can be called several times before it is used. In general, I found that although the **prevWin** variable provided all the functionality to navigate the windows created for a Tab user interface, it was awkward to do so.

In contrast, the leave command for toplevels provides a mechanism to undo the prior raise command on the toplevel window. For this purpose, the implementation of the raise command in Tab-Tk has to record which window was active when a toplevel gets raised. Providing the raise command with a dual provides exactly the functionality that is required to move between screens easily. Coming back to the above example, **exit_calculator** should execute **leave .calculator**. Figure 6 illustrates a generalization of this functionality. The use of the leave command is very intuitive, the leave command returns to the point from which the previous raise for the window was executed, it does not require any additional state saving in the implementation and therefore is much more effective. The only disadvantage of the leave command is that it adds functionality to Tk that is not available otherwise.

A Text Widget

The text widget for the Tab is similar to the Tk widget created with the **text** command. This widget provides a scrollable interface for reading and editing text. The main difference between the Tk text widget and the one provided for the PARCTAB is that a Tab text widget has two modes: reading and editing. When a text widget is first placed in a toplevel window it is in reading mode, and the TOP and BOTTOM buttons can be used to scroll through the text. In the reading mode, the stylus is used to position a cursor in the text and change the widget into editing mode. From then on the user edits the text using unistrokes [GR93], which is described below. Pressing the MIDDLE button exits the editing mode, returning to reading mode. Therefore, to reposition the cursor, one must exit editing mode by pressing the MIDDLE button, and then use the stylus in reading mode to select a new cursor position. Figure 7 shows how a text widget looks both in reading and editing mode. The second view of the screen shows the widget in editing mode with the cursor located right after the word "tiger", which has replaced the word "lion" from the first view.

Unistrokes is a writing method that associates a single pen-stroke shape with every letter in the English alphabet. Consequently, a letter is written by positioning the stylus on the Tab screen, drawing the shape for the desired letter, and then releasing the pen from the screen. Character

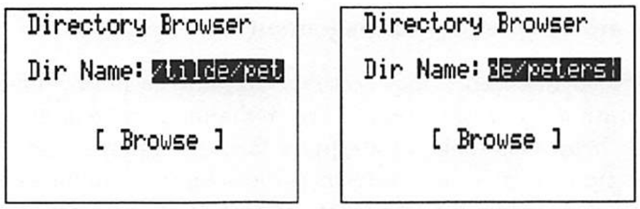


```
"We should look for
someone to eat and d
rink with before
 looking for somethi
ng to eat and drink,
 for dining alone
 is leading the life
 of a lion or wolf"
```

```
"We should look for
someone to eat and d
rink with before
 looking for somethi
ng to eat and drink,
 for dining alone
 is leading the life
 of a tiger¦ or wolf
```

```
text .browserfile.text
set contents [exec cat quote.txt]
.browserfile.text insert 1.0 $contents
place .browserfile.text -x 0 -y 0
```

Figure 7: Example of the text widget use.



```
toplevel .browserdata
label .browserdata.t  -text "Directory Browser"
entry .browserdata.e  -width 10
.browserdata.e insert 0 "/tilde/petersen"
label .browserdata.l  -text "Dir Name:"
button .browserdata.b -text {[ Browse ]} -command {
    set name [.browserdata.e get]
    enterdir 0 $name  # show the contents of the file or directory
}
```

Figure 8: Example of the entry widget use.

recognition is much simpler with the unistroke technology, and also more accurate. Although a user has to learn a new "alphabet" to use unistrokes, the experience at PARC has been that users become familiar with the technique easily. This can be attributed to two reasons: first, the symbols are fairly intuitive, and second, the unistroke shape is not rendered on the screen, instead the corresponding character is displayed at the cursor location as soon as the symbol has been terminated, this removes the focus from the unistroke symbol itself.

Like menus and toplevels, a text widget placed in a frame occupies the entire PARCTAB screen. To exit from the text widget, the MIDDLE button is pressed, which executes a **place forget** for the widget. Text widgets therefore need to be replaced each time the programmer desires the widget to be in the foreground.

In programming some of the user interfaces for the Tab, the need for a text entry widget that does not occupy the whole screen became evident. Interfaces that require input of a few words from the user, for example the name of a file, were somewhat awkward to implement for the Tab when only the widget created with the text command was supported. Therefore the widget provided by the **entry** command in Tk, which allows the user to edit an one-line text string, was added to Tab-Tk.

On the other hand, there is no real need for a widget similar to the one created with the **message** command, which is used to display multiline strings. The text widget is sufficient for this functionality, since it can be created with an empty binding for the stylus, thereby disallowing text editing.

The Entry Widget

The entry widget for the PARCTAB also has two modes: reading and editing. Initially the widget is in reading mode and looks like a text label in reverse video. The stylus is used to position a cursor in the entry and thereby changes the widget to editing mode. From then on the user edits the string by using unistrokes. Like for the text widget, the MIDDLE button exits editing mode, returning to reading mode. The TOP and BOTTOM buttons are bound to move the cursor left and right over the string.

When an entry widget is in editing mode, the entire surface of the Tab screen is used to write the unistroke symbols, which means that any other widget displayed on the screen at the same time will be inactive. The alternative would have been to allow the symbols to be drawn only in the area occupied by the entry widget. However, this solution is undesirable because the height of the area available to write the characters would have been extremely small.

Figure 8 shows an example use of an entry widget. Again the figure provides two views of the widget, the first illustrates an entry in reading mode and the second shows the same entry in editing mode.

Other Widgets and Geometry Management

The widgets described in the previous sections contain the major differences between Tab-Tk and the Tk implementation for workstations. The remaining widgets are fairly similar. For example, buttons and labels behave as their counterparts for X11. Frames also behave like those in X11, but are used much less frequently for user interfaces on the Tab, than for user interfaces for workstations. This difference can be attributed to two factors: first, and most important, the limited screen space available on the Tab considerably restricts the number of elements that can be displayed, which in turn reduces the need for elaborate grouping of widgets; second, the current implementation only provides the **place** command for geometry management, so widgets are mostly placed in absolute positions on the screen.

The decision to implement only the **placer** was a practical one. The placer provides a mechanism for simple fixed widget placement, limited to absolute and relative positions within the master. This limited capability makes row and column placement much less flexible than with the **packer**, the most commonly used geometry manager in Tk. However, the size of the PARCTAB screen imposes many limits on widget placement, hence making the sophisticated mechanisms provided by the packer seem extravagant. Furthermore, in practice, the placer has proven to be sufficient for the applications that have been developed for the Tab.

The last difference between the Tab-Tk widgets and their X11 counterparts, worth mentioning, is that the border and relief options are not implemented for the Tab-Tk widgets. Because the resolution of the Tab display is very low, 128×64 pixels, borders and reliefs would have consumed screen space that can otherwise be used to display more widgets. It was my decision in the design of Tab-Tk to favor the number of widgets that could be placed in a window over the sophistication of the widget's looks.

Experience with UI Reusability

One of the goals of using a scripting language like Tcl/Tk for Tab user interface programming was to exploit the potential of interface reusability across platforms. In other words, an interface designed in Tcl/Tk for the Tab should be usable on any other platform that provides a Tcl/Tk implementation.

Figure 9 shows the same Tcl/Tk script executing on both the PARCTAB and under X11. Picture 1 in the figure shows the main menu of the Tab application suite as displayed on the PARCTAB screen. The same menu posted on the leftmost side of the X11 execution is shown in picture 4. Notice that there are more elements visible in the menu displayed on the workstation than are on the Tab. The last three entries can be reached on the Tab interface by scrolling the menu with the external buttons. A small arrowhead in the lower right corner of picture 1 indicates that the menu contains more elements than shown on the screen.

Picture 2 in Figure 9 shows how a cascading menu is displayed on the PARCTAB. This submenu is reached by clicking on the movie camera icon of the main menu (picture 1) with the stylus. In picture 4 the behavior of the same cascading menu is shown for the workstation. The obvious difference between the two is that the whole cascading menu path is visible on the workstation, while on the PARCTAB only one menu is visible at the time. However, since menu navigation is hierarchical, not being able to see the outer menus on the Tab has not proven to be a problem. People tend to remember which steps they have taken during the menu navigation fairly well, thereby having the correct expectation of which menu is going to be posted when a submenu is unposted.

The toplevel window shown in picture 3 contains help information that can be reached by clicking on the "i" icon of the main menu. Again, the same information is displayed on the workstation in the toplevel window named "help". Notice, however, that on the workstation other toplevel windows are visible at the same time. While on the PARCTAB the stack model of toplevel windows only permits one window to be visible at a time, on the workstation all created toplevels can be seen. Furthermore, using the **raise** command to alternate between toplevel windows forces all the windows to be mapped in the workstation environment, which can make the display very busy and the interfaces somewhat cumbersome.

In hindsight, in order to maintain uniformity across platforms, it might have been better to have the window manager commands **wm deiconify window** and **wm iconify/withdraw window** to specify which toplevel occupies the PARCTAB screen. This change would not affect the look and feel on the Tab, since still only one toplevel would be visible at a time. However, this approach would make user interfaces written for the Tab behave more appropriately on a workstation.

The only other reason that user interfaces designed for the PARCTAB appear incorrect on the workstation is that the placement of labels and buttons with text images. Since font sizes vary across platforms, the absolute placement of these widgets produces different results on different displays. This is the one case that clearly favors the use of the packer as the geometry manager, because it allows widgets to be placed in relation to other widgets.

In general, though, the user interfaces designed for the PARCTAB can be used without changes for workstations. This property has been very useful, particularly for debugging purposes.

Generation of Applications

A primary reason for integrating Tcl/Tk into the PARCTAB system was to support quick prototyping and implementation of applications for the device. The experience in the laboratory showed that programming applications using the Modula-3 TabBits library of widgets was not attractive for most people. Programmers needed to be aware of too many details regarding the device in order to generate even the simplest of interfaces. In addition, applications implemented for the PARCTAB using this library cannot be directly used on other platforms and are difficult to customize. The rest of this section focuses on existing alternatives to implement and customize applications for the Tab and other PDAs.

A different approach was used for the memory prosthesis project at EuroPARC [LBC+94], in which the PARCTAB screen was made to mimic a section of a Macintosh display. With this approach the Tab displays a copy of the bitmap corresponding to the dedicated section of the Macintosh screen, and the events from the Tab are forwarded to the Mac, which performs the appropriate actions locally.

Although this approach relieves the programmer from learning the intricacies of the PARCTAB programming environment, the interfaces have to be created using a programming environment for a platform that is different than the target, therefore lacking support for managing limitations such as the available display area. Furthermore, an interface designed in this fashion for the PARCTAB could not be directly ported to many other types of devices, since the the implementation depends heavily on the PARCTAB system architecture and the programming platform. Another disadvantage of this approach is that a portion of the Macintosh screen is needlessly occupied by an application that has nothing to do with the desktop environment provided by the machine. In summary, this approach only relieves the implementor from the learning curve associated with the PARCTAB infrastructure, but retains all the disadvantages of implementing the applications with a platform specific windowing library.

A general purpose windowing toolkit relieves the programmer from most of the problems mentioned previously. The toolkit implementation effectively exploits the characteristics of a specific device, but the application implementor will not have to be aware of the low level details of the target device. In fact, our experience demonstrated that within three months of deploying the Tcl/Tk infrastructure for the Tab system, several people had implemented applications such as a remote control for a conference video camera with very little effort. The general comment was "... and I was surprised how easy it was to use." Moreover, people have customized their PARCTAB environments by making small changes to their Tcl/Tk initialization scripts.

Customization of user interfaces is nearly impossible for applications coded directly in a low-level native programming environment. For PDAs, which in general have small displays, the lack of support for customization can be a severe limitation. People have different preferences for their working environments, and customization can be an important factor in how extensively a system is used.

Some system designs for PDAs provide drag and drop interfaces for customization. For example, General Magic's Magic Cap includes "drawers and folders in file cabinets" with widgets that provide predefined add-on functionalities, such as notifications when a certain event occurs, mail message formats, labeling of calendar entries or mail messages with urgent marks, logging of messages that are received from specific users, and others. However, the drag and drop approach to customization assumes a minimum of screen real estate. It would be impossible to provide this type of interface for the Tab, since there is not enough area on the screen to display both the folder of possible add-on functionalities and the window of the application that is being customized.

Tcl/Tk provides a straightforward means of customization. Commands and widgets can be redefined by either modifying the source scripts or by sending redefinitions of the commands and widgets to the application's interpreters. Because scripts are interpreted, these two alternatives allow applications to be changed almost instantaneously.

Although there is a definite need for a low-level library of windowing widgets such as TabBits or libX as the basis for window systems, the user interface implementor should be spared that level of detail. Tcl/Tk has provided a very effective mechanism for easy application design and customization for the PARCTAB system. The Tk windowing toolkit allows fast user interface prototyping, and its capacity for being embedded in applications is being used to link the PARCTAB user interface with other applications. Some of the new applications that were enabled for the Tab by providing the Tcl/Tk infrastructure include:

- A remote user interface to ppres [Per92], a presentation manager implemented in Tcl/Tk. The TOP and BOTTOM buttons are used to select the previous or next slide being presented, and the stylus is used to move the cursor within the slides remotely.
- A remote control for a Canon video camera installed in the laboratory's conference room. The remote control interface allows the camera to rotate and tilt on its base, and the focus to be zoomed in and out. The user interface on the PARCTAB communicates with an application that sends the appropriate commands to the camera over a serial link.
- A reminder application that uses the location-aware capabilities of the Tab to allow applications to react dynamically to the locations in which they are situated. The user associates reminders with rooms, which pop up as messages on the Tab screen when the user enters a room for which a reminder has been specified.
- A remote postit note editor, which enables users who are away from their office to edit notes, which are then send to the user's workstation and displayed as postit-notes on the workstation's screen.

Conclusions

The project of porting Tcl/Tk to the PARCTAB system was primarily motivated by the observation that rapid prototyping and implementation of applications for the PDA was crucial to extend the use of the device. Although low-level libraries of user interface widgets are necessary building blocks for window systems, the user interface implementor should not have to be aware of that level of detail.

Tcl/Tk has been used effectively in the last few months in our laboratory to create and customize applications for the Tab. Tab-Tk has allowed rapid user interface prototyping. Furthermore, the capacity of Tcl for being embedded in applications has been very important to link remote applications, such as presentation managers and video camera controls, to the PARCTAB user interfaces. This experiment has further demonstrated the impact of being able to use the PARCTAB in combination with the other computers in our environment, rather than as a stand-alone device. We are currently in the process of exploring more applications for which the Tab can be used as a remote control.

One of the key advantages of using Tcl/Tk rather than a platform-specific scripting language for user interface design and communication is that interfaces designed in Tcl/Tk for the Tab can be reused on any other platform that provides a Tcl/Tk implementation. We experimented with the reusability of UIs and learned that it can be very powerful. We also learned that the windowing toolkit implemented for the device needs to provide all the mechanisms that support device-independent programming. In Tab-Tk, for example, the packer would have been more adequate to place widgets with text images than the placer, even though most of the packer's other functionality could seem extravagant.

Tcl/Tk provides an effective way of customizing interfaces and applications. Commands and widgets can be redefined easily and without paying large compilation or application startup overheads. For devices such as the PARCTAB, which have small displays, being able to customize applications in this way is very important, since "drag and drop" customization techniques require more screen real estate than available.

The design of Tk for the Tab was mainly driven by the size of the PARCTAB's display. Most decisions made while porting Tk to the Tab focused on maintaining the natural look and feel of the Tk widgets and at the same time exploiting the small area of the display as effectively as possible.

Although the system architecture of the PARCTAB is somewhat different than that of most other PDAs, the experience obtained with this work applies equally to any PDA, particularly with respect to the application generation, customization, and interoperability issues. When compared to the Newton and HP100, the size of the PARCTAB display is small. However, we expect other ubiquitous computing devices to emerge that will have equally small or smaller displays and will therefore benefit from the experiences learned from the Tab.

Acknowledgements

I would like to thank Brent Welch for the hours of discussions concerning everything from quoting to concepts in Tcl/Tk; Bill Schilit for the time explaining, debugging, maintaining, and extending the Tab system; Norman Adams for supporting the Tab system and pointing out interesting race conditions; Roy Want for supporting and encouraging my plans for the remote control project; and Mark Weiser for his support for ideas in ubiquitous computing.

References

[EBG+92] Scott Elrod, Richard Bruce, Rich Gold, David Goldberg, Frank Halasz, William Janssen, David Lee, Kim McCall, Elin Pedersen, Ken Pier, John Tang, and Brent Welch. Liveboard: a large interactive display supporting group meetings, presentations, and remote collaboration. In *Proceedings of CHI '92. Human Factors in Computing Systems*, pages 599–607, May 1992.

[GM994] General Magic Informatin Binder. General Magic Inc., 1994.

[GR93] David Goldberg and Cate Richardson. Touch-Typing with a Stylus. In *Proceedings of the INTERCHI'93, Conference on Human Factors in Computing Systems*, pages 80–87, April 1993.

[Kim94] James Kim. Wear Datebook on your Wrist. USA TODAY, June 22 1994.

[LBC+94] Mik Lamming, Peter Brown, Kathleen Carter, Margery Eldridge, Mike Flynn, Gifford Louie, Peter Robinson, and Abigail Sellen. The Design of a Human Memory Prosthesis. *To appear in Computer Journal*, 1994.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Per92] Perspecta Presents. User's Guide. Perspecta Software Inc., 1992.

[PL92] Ken Pier and James A. Landay. Issues for Location-Independent Interfaces. Technical Report ISTL-92-4, Xerox Palo Alto Research Center, December 1992.

[SAG+93] Bill N. Schilit, Norman Adams, Rich Gold, Michael M. Tso, and Roy Want. The ParcTab Mobile Computing System. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 34–39, October 1993.

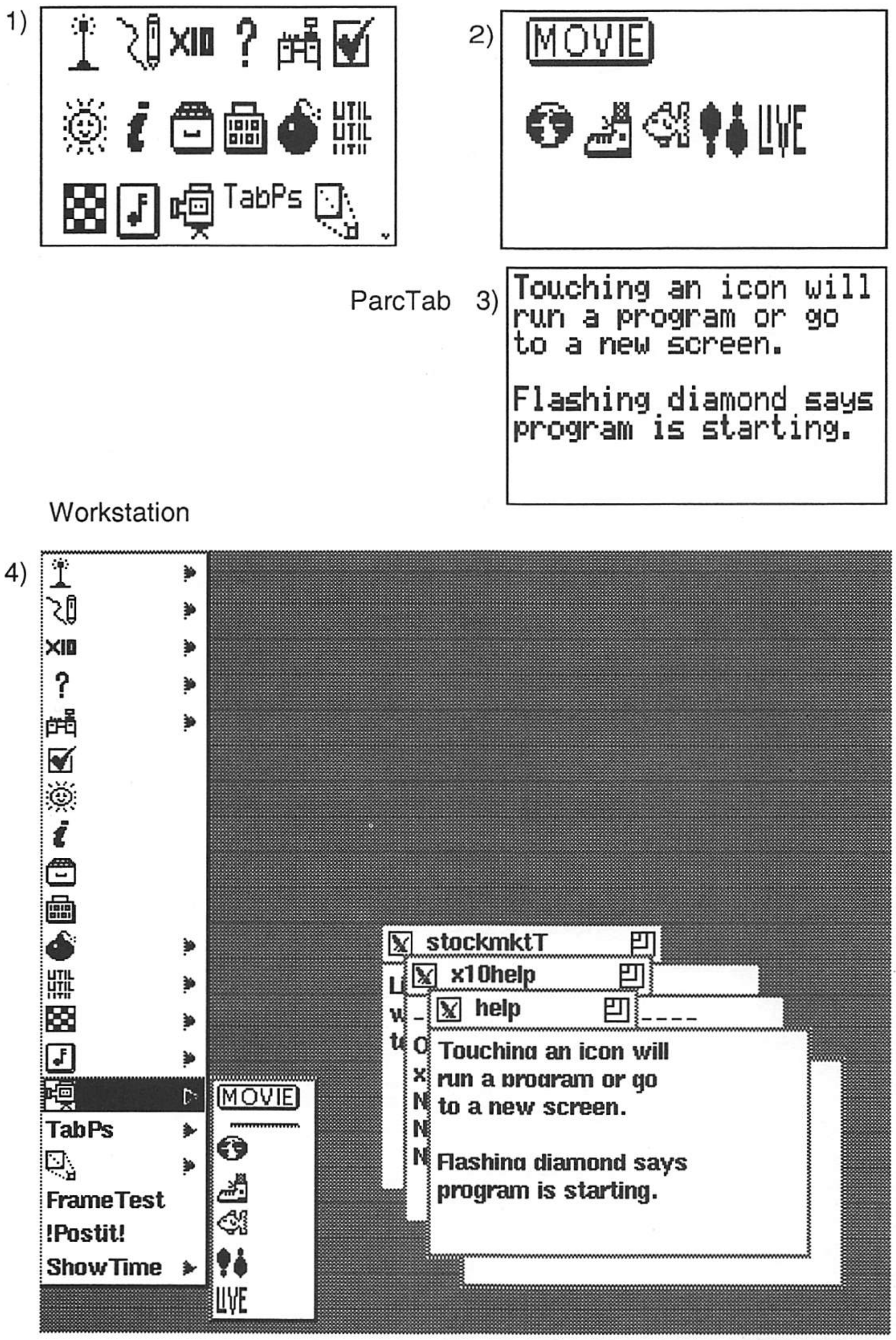
[Smi94] Walter R. Smith. The Newton Application Architecture. In *Proceedings of the 1994 IEEE COMPCON*, February 1994.

[Ude94] Jon Udell. COMPONENTWARE. *Byte*, 19(5):46–56, May 1994.

[Wei91] Mark Weiser. The Computer of the 21st Century. *Scientific American*, pages 94–104, September 1991.

Author information

Karin Petersen has been a Member of the Technical Staff in the Computer Science Laboratory at Xerox PARC since September of 1993. Prior to joining PARC, Karin received an Engineering degree in Computer Science from the Simón Bolívar University (Venezuela) in 1988, and a Ph.D. in Computer Science from Princeton University in 1993. Her research focuses on distributed and parallel systems, performance evaluation, and user interfaces for mobile and distributed applications.



Touching an icon will run a program or go to a new screen.

Flashing diamond says program is starting.

Figure 9: Same UI on the PARCTAB and a workstation

Using Tcl to Control a Computer-Participative Multimedia Programming Environment [*]

Christopher J. Lindblad [†]

Telemedia Networks and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

This paper describes how the VuSystem, a programming environment for the development of computer-participative multimedia applications, is controlled through Tcl scripts. In the VuSystem, networks of in-band media-processing modules are created and controlled by interpreted out-of-band Tcl scripts through object commands and callbacks. Tcl's extensibility, simple type system, efficient interface to C, and introspective capabilities are used by the VuSystem to produce a highly dynamic and capable media processing system.

1 Introduction

The *VuSystem* [2, 4] is a programming environment for the development of computer-participative multimedia applications. It is designed to run on high performance computer systems not specifically designed for the manipulation of digital video. The system is unique in that it combines the programming techniques of visualization systems and the temporal sensitivity of traditional computer-mediated multimedia systems.

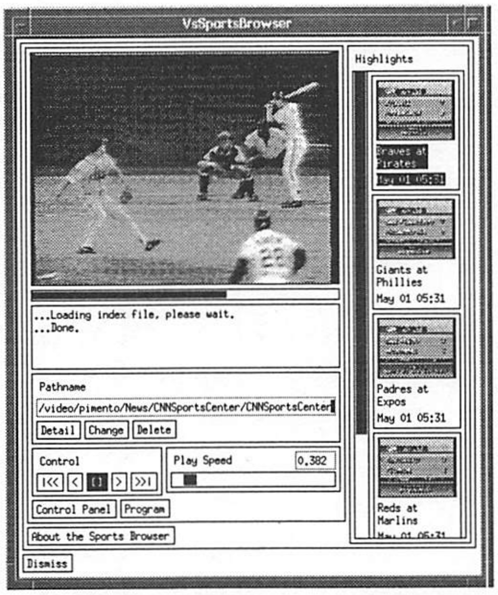
VuSystem code is split into two partitions: one which does traditional *out-of-band* processing and one which does *in-band* processing. Out-of-band processing is that processing which performs event handling and other higher-order functions of a program. In-band processing is the processing performed on every video frame and audio fragment.

This architecture differs from that of traditional multimedia toolkits that either provide a low level language interface to a library of primitives [12, 13], or provide a high level language system with limited extensibility [11]. In the VuSystem, in-band code can be written in a low level language and can be optimized for performance, while out-of-band code can be written in a high level language and optimized for programmability, usability and extensibility.

In the VuSystem, the in-band processing partition is arranged into processing *modules* which logically pass dynamically-typed data *payloads* though input and output *ports*. These in-band modules can be classified by the number of input and output ports they possess. The most common module classifications are sources, with no input ports and one output port; sinks, with one input port and no output ports; and filters with one input port and one output port. In-band media-processing code is more elaborate and extensible in the VuSystem than in traditional computer-mediated multimedia systems because VuSystem applications perform more analysis of their input media data.

[*]This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the United States Air Force (AFSC, Rome Laboratory) under contract No. F30602-92-C-0019, and by a grant from Nynex.

[†]The author can be reached at: MIT Laboratory for Computer Science, Room 504, 545 Technology Square, Cambridge, MA 02139; Tel: +1 617 253 6042; Email: cjl@lcs.mit.edu.



```
...Loading index file, please wait.
...Done.
```

Pathname: /video/pimento/News/CNNSportsCenter/CNNSportsCenter

Figure 1: The Sports Highlight Browser, written with the VuSystem. The smaller panes on the right contain scoreboard graphics reporting final scores of sporting events. Clicking on one shows the video highlights of the event in the larger pane.

The nature of out-of-band processing is very different from in-band processing. For the out-of-band code, a programming system can be chosen that can handle user interfaces and other event-driven program functions well. When designing the out-of-band partition, programmability is more important than performance. For maximum ease of application development, the out-of-band partition of VuSystem is programmed in an interpreted scripting language. Application code written in this scripting language is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application.

The scripting language used in the VuSystem is the Tool Command Language, or Tcl [8]. Application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application. In-band modules are manipulated with *object commands*, and in-band events are handled with *callbacks*.

The VuSystem is implemented on Unix workstations as a shell-like program that interprets an extended version of Tcl. In-band modules are implemented as C++ classes and are linked into the shell. Simple applications that use the default set of in-band modules are written as Tcl scripts. More complicated applications require linking in additional in-band modules to the default set.

Several computer-participative multimedia applications have been developed with the VuSystem [3]. *The Room Monitor* and *The Sports Highlight Browser* are two representative examples. The Room Monitor processes video from a stationary camera to determine if a room is occupied or empty, and records only when activity is detected above some threshold, producing a series of video clips that summarize the activity in the room. The Sports Highlight Browser (Figure 1) provides access to a sporting news telecast that has been automatically segmented into a set of video clips, each of which represents highlights of a particular sporting event.

VuSystem programs have a *media-flow* architecture: code that directly processes temporally sensitive data is divided into processing *modules* arranged in data processing *pipelines*. This architecture is similar to that of some visualization systems [16, 18], but is unique in that all data is held in dynamically-typed time-stamped *payloads*, and programs can be reconfigured while they run. Timestamps allow for media synchronization. Dynamic typing and reconfiguration allows programs to change their behavior based on the data being fed to them.

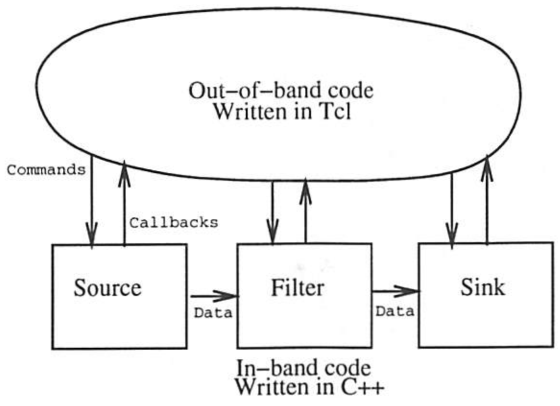


Figure 2: The structure of VuSystem applications. Out-of-band code written in Tcl creates an controls in-band media processing modules written in C++.

2 The Tool Command Language

The Tool Command Language is an excellent programming language for assembling modules into flexible applications. Tcl is designed as a simple but extensible command language. Its syntax is concise enough that simple Tcl commands can just be typed in, but it is programmable and powerful enough that most of the control logic of a large application can be written in it. It has a simple and efficient interpreter, and a simple interface to C.

Tcl syntax is similar to that of Unix shells, but it has additional Lisp-like constructs: Tcl uses curly braces to group elements, square brackets to invoke command substitution, and dollar signs to invoke variable substitution.

Tcl's Simple Type System

In Tcl, all commands and values are strings. There is no other data type. It has no native representation of numbers or lists. All data is in the form of character strings. Even Tcl commands themselves are strings. Since all data in the Tcl interpreter are strings, the embedding interface is simplified. It is easy to pass data between the Tcl interpreter and C code in an application. The C code need only be able to convert internal objects to and from strings. No library of converters between multiple representations is required.

Data types that can be easily represented in string form are quite natural to use within Tcl. For example, numbers can be easily converted to and from strings using standard mechanisms, and lists can be easily represented as strings, using curly braces for grouping. Data types too complex to be efficiently converted to and from strings can be represented in Tcl with *handles* or *object commands*.

Representing Complex Objects in Tcl

Some objects of data types too complex to be efficiently converted to and from strings can be represented with string *handles*. Primitive commands that manipulate these objects use standard methods to convert string handles to objects, and from objects to string handles. A good example of objects that are represented by string handles are open files. The standard Tcl library provides commands to open, close, read and write files. These primitives use file *handles*, short strings that can be converted to and from file descriptors.

The most powerful way to represent objects too complex to be efficiently converted to and from strings is through *object commands*. In this approach, for each object of a complicated data type, a unique Tcl Command is registered in the interpreter. Operations on an object are performed by invoking its Tcl command, with the first argument to the command specifying the operation, and

Source	Function
VsSunAudioSource	Interfaces directly to the audio device on Sun workstations.
VsAudioFileSource	Interfaces to an AudioFile [10] server typically running on a Digital Alpha workstation.
VsSunVfcSource	Interfaces to the VideoPix video capture card on Sun workstations.
VsXVideoSource	Interfaces to video display adapters that have video capture capability through the XVideo X extension.
VsVidboardSource	Interfaces to the Vidboard [9], a LAN-based video capture subsystem.
VsFileSource	Interfaces to files stored using the native VuSystem file format.
VsQtimeSource	Interfaces to Quicktime [12] files.
VsMpegSource	Interfaces to MPEG [15] files.
VsCaptionSource	Interfaces to a closed-caption decoder through a serial line.
VsExternalSource	Assembles separate image files into a sequence of video frames.

Table 1: VuSystem sources.

Sink	Function
VsSunAudioSink	Interfaces directly to the audio device on Sun workstations.
VsAudioFileSink	Interfaces to an AudioFile [10] server typically running on a Digital Alpha workstation.
VsWindowSink	Interfaces to video display adapters through the X Window System.
VsFileSink	Interfaces to files stored using the native VuSystem file format.
VsQtimeSink	Interfaces to Quicktime [12] files.
VsExternalSink	Causes a sequence of video frames to be written to several image files.

Table 2: VuSystem sinks.

the rest of the arguments specifying the arguments to the operation. The Tk graphical user interface toolkit uses object commands to manipulate widgets [8].

Object commands are used by the VuSystem to manipulate media processing modules. Each module is manipulated with its own object command. Each object command has several subcommands that allow the state of its object to be queried and changed. VuSystem object commands are constructed using Object Tcl [5], a dynamic object-oriented extension to Tcl that was developed for this purpose.

3 Manipulating Modules

VuSystem media processing modules are created in Tcl with *class* commands and manipulated with *object* commands. A class command is defined for each type of module that can be created, and an object command exists for each module created. For example, the VsWindowSink Tcl command creates a VsWindowSink module, and installs a new command in the Tcl interpreter to control

Filter	Function
VsJpegC/VsJpegD	Compresses/decompresses video frames using the JPEG [14] video compression standard.
VsCCCC/VsCCCD	Compresses/decompresses 8-bit color video frames using a color cell compression algorithm.
VsQRLC/VsQRLD	Compresses/decompresses grayscale video frames using a simple run-length coding scheme.
VsColor24to8/VsColor8to24	Converts between 24-bit color and 8-bit color video frames.
VsColor24toGray	Converts 8-bit color to grayscale video frames.
VsChannelSelect	Only passes payloads with certain channel identifiers.
VsChannelSet	Sets channel identifiers for VsChannelSelect.
VsReTime	Changes the starting-time and duration on payloads.
VsStepper	Allows explicit lockstep scripting control of each video frame passed.
VsWait	Waits for VsFinish payloads.
VsRateMeter	Measures the flow rate of payloads through it.
VvEdge	Performs edge processing on video frames.
VvThresh	Performs thresholding on pixel values.
VvHistogram	Converts a video frame into a pixel value histogram.
VsTcpClient	Interfaces to the client end of a TCP connection.
VsTcpServer	Interfaces to the server end of a TCP connection.

Table 3: VuSystem filters.

Module	Function
VsDup	Copies payloads to provide identical output sequences.
VsTap	Copies payloads to provide identical output sequences.
VsDeMux	Splits its input sequence based on payload channel number.
VsMerge	Merges multiple payload sequences into a single sequence.
VsMux	Merges multiple payload sequences into a single sequence.
VsOrderedMux	Merges multiple payload sequences into a single sequence, but ensures that timestamps in the output sequence are always monotonically increasing.
VsFade	Provides a fade effect between two sequences of video frames.

Table 4: VuSystem modules with more than one input and output.

the module. The VsWindowSink Tcl command takes as its first argument the name of the object command to create. The rest of the arguments to the class command are parameters for the new module.

Module Types

VuSystem modules are best categorized by how many input and output ports they have. A module is either a *source*, a *sink*, a *filter*, or some *other* module.

Modules with no input ports and one output port are called *Sources* because they appear to the VuSystem to source data. Sources typically interface to media capture devices or media storage systems. *Audio* sources interface to audio capture hardware. *Video* sources interface to video capture hardware. *File* sources interface to files. More exotic sources exist as well. Some sources provided with the VuSystem are listed in Table 1.

Modules with one input port and no output ports are called *Sinks* because they appear to the VuSystem to sink data. Sinks typically interface to media playback devices or to media storage

Subcommand	Modules	Function
callback	All Modules	Get and set the callback.
children	All Modules	List all children in the structural hierarchy.
color	Video Sources	Toggle between color and grayscale.
destroy	All Modules	Destroy module and all its children.
frameRate	Video Sources	Get and set the video frame rate to sample.
gain	Audio Sources and Sinks	Get and set the gain.
hue	Video Sources	Get and set the hue.
pathname	File Sources and Sinks	Get and set the pathname.
port	Many Sinks	Get and set the hardware output port.
port	Many Sources	Get and set the hardware input port.
scale	Video Sources	Get and set the video frame size.
start	All Modules	Start module and all its children.
stop	All Modules	Stop module and all its children.

Table 5: VuSystem object subcommands.

Condition	Modules	Indication
caption	VsWindowSink, VsCaptionSink	A caption was received.
compressRatio	Compression Filters	Filter is achieving this compression ratio
detect	VsPayloadDetect	A payload of the specified type has been detected.
done	Effects Modules	The effect has completed.
rate	VsRateMeter	Payloads are passing at this rate.
sinkFinish	All Sinks	A VsFinish payload was received.
sinkStop	All Sinks	A VsFinish payload was received while in stopping mode.
solve	VsPuzzle	The puzzle has been solved.
sourceEnd	All Sources	The end of the source has been reached.
stepDone	VsStepper	A payload has been sent.
waiting	VsWait	A change of waiting status has occurred.

Table 6: VuSystem callback conditions.

systems. *Audio* sinks interface to audio playback hardware. *Video* sinks interface to video playback hardware. *File* sinks interface to files. Some sinks provided with the VuSystem are listed in Table 2.

Modules with one input port and one output port are called *filters* because they are typically used to perform signal processing operations on the data flowing through them. *Compression* filters compress or de-compress video frames. *Pixel format conversion* filters convert the format video frames. *Descriptor* filters perform operations on the descriptors of payloads. *Visual* processing filters perform various functions on video data. Some filters provided with the VuSystem are listed in Table 3.

Modules with more than one input or output ports provide mechanisms for splitting and merging payload sequences. Some modules with one input port and many output ports split a single timestamped sequence of payloads into multiple sequences. Some modules with many input ports and one output port merge multiple sequences into a single sequence. Others combine payload sequences from two input ports to one output payload sequence, operating on the contents of the data. Table 4 lists some VuSystem modules with more than one input and output port.

4 Communication Between In-Band and Out-Of-Band

In the VuSystem, out-of-band Tcl scripts and in-band C++ modules communicate through *object commands* and *callbacks*:

- Out-of-band code is able to create and destroy in-band modules, query the state of in-band modules, and give commands to in-band modules, all through special Tcl *object commands* defined for each in-band module and port.
- In-band media-processing code signals out-of-band Tcl code whenever an appropriate in-band event occurs through Tcl *callbacks*.

Object commands are always completed in the in-band partition *synchronously* with the out-of-band requester: object commands execute immediately and completely when called from out-of-band scripts. In contrast, because of the time-critical nature of in-band code, it is unacceptable for in-band code to wait for a response from out-of-band code. Tcl *callbacks* are executed in the out-of-band partition *asynchronously* with the in-band partition: callbacks are only queued for execution when invoked from in-band code. Later, the VuSystem scheduler actually executes them. Since out-of-band callbacks do not execute immediately when they have been signalled by in-band code, they are only used to signal events to the out-of-band code, and cannot return values to their in-band signallers. Any in-band changes made by an out-of-band callback are performed through object commands.

Object Subcommands

Each module object command has a set of *subcommands* that vary according to type of the module. The internal state of the module can be queried and changed with some of these subcommands. For example, the VsVidboardSource module has a `port` subcommand that is used to control from which input port it captures video. Table 5 shows a list of typical module subcommands.

Subcommands are implemented as normal Tcl command procedures, whose client data argument by default is a pointer to the associated module. Subcommands are declared *friend* procedures to the module class, so they may manipulate private members of a module.

Callbacks

In-band modules process continuous sequences of media data, while out-of-band Tcl control processing deals with events. Sometimes, out-of-band Tcl code in an application should be executed when an in-band event occurs. In this case, an in-band module would call a Tcl *callback*. The VuSystem provides a facility for each module to have a callback, which can be called whenever a specific event occurs during in-band processing. For example, the VsFileSource module calls its callback when it reaches end-of-file.

Callbacks are defined in Tcl. Typically the Tcl application programmer uses a name of a Tcl procedure as the callback command. The Tcl procedure looks at its arguments to determine what event has occurred. Tcl callback commands are installed with the `callback` subcommand. Each module can have only one callback installed at a time. If a module can signal more than one type of event, it supplies keyword arguments to the callback command, so the command can determine which event occurred. Table 6 shows a list of typical module callbacks.

Example

Figure 3 shows how a Tcl application programmer might make use of a simple file source callback that indicates end-of-file. This example code provides the automatic switching of the file source from the file first.uv to the file second.uv when end-of-file is encountered on the first file.

The `sourceCallback` procedure takes a keyword argument list in its args parameter. It extracts any `-sourceEnd` keyword parameter with the `keyarg` command, defaulting to 0. If `sourceEnd` is nonzero, `sourceCallback` changes the file for the source module using the `pathname` subcommand for

```
proc sourceCallback {args} {
  set sourceEnd [keyarg -sourceEnd $args 0]
  if $sourceEnd {
    vs.source pathname "second.uv"
    vs.source callback ""
  }
}

SimpleFileSource vs.source \
  -pathname "first.uv" \
  -callback "sourceCallback"
```

Figure 3: How a simple file source callback might be used.

the module. This will cause the source module to start on the file second.uv. The **sourceCallback** procedure also clears the callback for the source module using the **callback** subcommand for the module so that when the end of second.uv is signalled, it does not run again.

After defining the **sourceCallback** procedure, a SimpleFileSource named **vs.source** is created, with its input file set to first.uv and its callback set to **sourceCallback**. When started, vs.source will read from first.uv and evaluate the Tcl command string "**sourceCallback -sourceEnd 1**" when it encounters end-of-file.

5 The VuSystem Application Shell

The VuSystem is implemented as a Unix *application shell*: it is program that interprets an extended version of Tcl. Linked into the program are all standard in-band modules, implemented as C++ classes. Tcl scripts implement simple applications that use the default set of in-band modules. By linking additional in-band modules into the application shell, more complicated applications can be constructed.

The application shell defines the interface between the primitive module and command developer and the application script developer. At the primitive level, the module developer creates new primitive VuSystem modules and primitive Tcl commands and links them into the VuSystem application shell. At the application level, the developer writes Tcl code that runs in the application shell.

6 Programming the Graphical User Interface

I implemented a Tcl interface to the X Window System Toolkit [17] and the Athena widget set for the graphical user-interface code. At the start of VuSystem development, I chose to use Xt and the Athena widget over the Tk widget set provided with the Tcl distribution, because at that time the Xt intrinsics and Athena widget set were more robust and complete, and also had built-in features for scheduling in large applications. Through the scheduling interface provided by the Xt intrinsics, I provided scheduling to the in-band modules. Today, the VuSystem could be changed to use Tk, since these capabilities are now provided by Tk.

The TclXt and TclXaw components of the VuSystem [4] provide Tcl programming interfaces to the standard X Window System Xt and Xaw libraries. These components enable the Tcl programmer to construct graphical user interfaces based on the Xt toolkit and the Athena widget set. TclXt and TclXaw use object commands to manipulate X displays, application contexts, events, widgets, and widget classes. Widget resources and other object state can be manipulated through subcommands to these object commands. They provide an interface to the Athena widget set that is similar to Tk, as follows:

- Widget instances are created by invoking a WidgetClass command. For example, to create a button, one uses the Command Tcl command, which creates a Command widget.

- Initial values for widget resources are provided as keyword arguments to the class command. The standard string conversion facilities provided by the Xt intrinsics convert the resource values from strings.
- Resources of existing widgets can be queried and changed through widget *sub-commands*. Just as for the initial values of these resources, the standard string conversion facilities provided by the Xt intrinsics convert the resource values to and from strings.
- *Callbacks* and *Translations* can be specified as Tcl commands. These commands are executed whenever the given input event occurs.

TclXt and TclXaw provide a powerful and complete interface to the Athena widget set. They make the entire interface to the Xt and Xaw libraries acessible to the Tcl programmer. With TclXt and TclXaw, there is no library interface available to the C programmer that is not also available to the Tcl programmer.

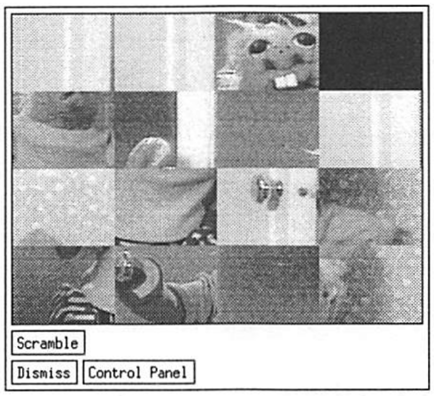


Figure 4: A video puzzle application.

7 Example Application Script

The following is a simple example application built out of a video source module, a video sink module, and a filter module. It implements a video version of a 16-square puzzle. It takes input from a camera or other video source, scrambles it, and then presents the output on a window. Figure 5 shows a block diagram of the in-band modules employed in this application.

The Tcl script fragment that configures the modules and starts them running is shown in Figure 6. This fragment is not complete, but captures the essence of the larger script.

This script first creates an instance of a VidboardSource module and names it `m.source`. Then the script creates an instance of the Puzzle module and names it `m.filter`. It also instructs this module instance to connect its input port to the output port of the `m.source` module instance. The output port of `m.source` was automatically named `m.source.output`.

Next, the script creates a WindowSink module, specifying with the `-window` option that the widget named `w.screen` is the window on the screen to use. The input port of the WindowSink module is also specified to be connected to the output port of the Puzzle module.

Finally the parent module instance named `m`, created before this script fragment was run, is given the `start` command, causing all its child module instances, those with name `m.`*whatever*, to start.

In addition to this script fragment, the puzzle application includes similar code to construct its user interface, and code to cause the `m.puzzle` module instance to reconfigure itself whenever the user moves a block in the puzzle.

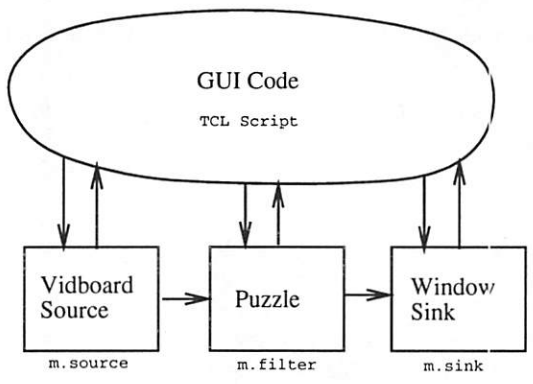


Figure 5: Block diagram of a video puzzle application.

```
VidboardSource m.source

Puzzle m.filter \
  -input "bind m.source.output"

WindowSink m.sink \
  -window w.screen \
  -input "bind m.filter.output"

m start
```

Figure 6: Tcl code from a video puzzle application.

8 Extensions to the VuSystem

A few projects are under way to extend the scope of the VuSystem. Work in progress includes a visual programming system for media computation and a distributed programming system for media applications.

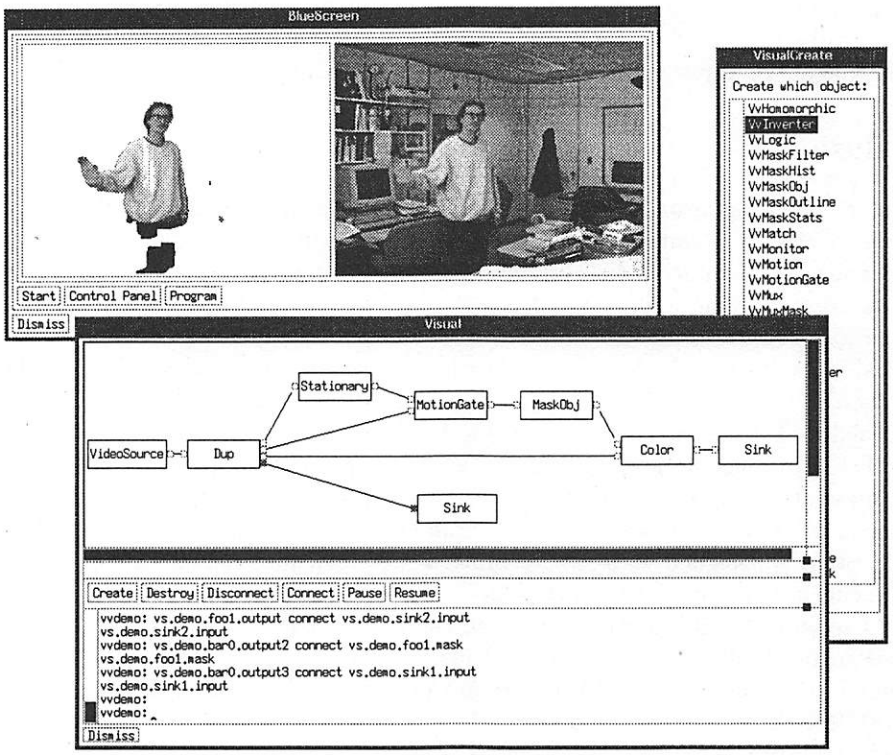
Interactive Programming

A visual programming system for application users is nearing completion. Users interact with the system through a flow graph representation of the running program to control its media processing component. A "flow graph" perspective emphasizes the computation that occurs, rather than a "hypermedia" perspective, which may view the media in terms of a database to be navigated. Flow graph, or dataflow, representations have been used with success in prior visual languages [18].

The visual environment is suited to tasks such as customization, rapid prototyping and experimentation, as well as more general program development. It provides a programming ability (rather than a limited set of configuration options) to users, allowing them to re-program previously developed applications. By embedding it in a toolkit, consistent user programming facilities are available in all derived applications.

From the user's point of view, the visual environment consists of a number of display windows (Figure 7). One window shows the media flow graph representation, the primary means of user programming. Another is a customization panel, detailing the individual options that may be selected for each module of the program's computation. Further objects allow interactions with the textual programming methods of the VuSystem. Description panels show the code fragment associated with a module, and an Interpreter window evaluates commands on demand.

The visual environment demonstrates the modularity, flexibility, and extensibility of Tcl and the VuSystem. Its implementation required only a few additions to the VuSystem core; it works from within the VuSystem; and it is mostly written in Tcl. The visual environment also demonstrates



```
vvdemo: vs.demo.foo1.output connect vs.demo.sink2.input
vs.demo.sink2.input
vvdemo: vs.demo.bar0.output2 connect vs.demo.foo1.mask
vs.demo.foo1.mask
vvdemo: vs.demo.bar0.output3 connect vs.demo.sink1.input
vs.demo.sink1.input
vvdemo:
vvdemo:
```

Figure 7: The visual programming interface to the VuSystem, manipulating some vision service modules. The program running is a smart blue-screen application, which separates the active foreground from the stationary backgound in a video sequence. The diagram is of the VuSystem modules that comprise the program.

the introspective capabilities of Tcl and of the VuSystem. The environment shows that VuSystem applications can examine and modify themselves while they run.

Distributed Programming with VuDP

Currently, all modules of a standard VuSystem program must execute in the same local environment on a single host. Applications split across the network must be realized as a set of co-operating programs, making them difficult to write. Using VuDP, a program may be constructed from modules that exist in different environments distributed across several hosts.

Under development, the VuSystem Distributed Programming (VuDP) extension will simplify the construction of applications whose processing is distributed across the network. Three advantages offered by the VuDP model are: transparent access to shared resources, the ability to divide applications across hosts, and the enabling of collaborative applications.

VuDP will support distributed programming through three mechanisms: a remote evaluation capability, an extensible set of exportable services, and a network based model for intra-application communication. The remote evaluation capability will provide a general means of creating remote modules in a suitable execution environment. The exportable services will work in an RPC-like manner to support common tasks. Finally, VuDP will support both in-band communication for media flow and out-of-band communication for control between modules at different sites.

VuDP will leverage off features of Tcl and the VuSystem that make distributed systems easy to implement. The remote evaluation component of VuDP will work by passing Tcl commands and values over reliable byte-stream TCP connections. Since all commands and values in Tcl are strings, linearization for this network transport is trivial. In addition, linearization code for in-band

VuSystem payloads has already been implemented in support of the native VuSystem file format. This support will also be used for the transport of in-band data over reliable byte-stream connections. Distributed VuSystem applications will be built using this mechanism.

9 Conclusion

The VuSystem is a programming environment for computer-participative multimedia applications. It is unique in that it combines the computational flexibility of visualization systems with the temporal sensitivity of multimedia systems. VuSystem applications are partitioned into *in-band* code that manipulates the audio or video data, and *out-of-band* code that performs higher-level event-driven functions.

The in-band partition of the VuSystem is structured as a reconfigurable directed graph of *modules* that logically pass timestamped *payloads* holding media data. The out-of-band partition of VuSystem is programmed in Tcl, an interpreted scripting language well suited to the task. Application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application. In-band modules are manipulated with Tcl *object commands*, and in-band events are handled with Tcl *callbacks*.

The VuSystem is designed to be as portable as possible, but includes support for some proprietary media capture and presentation interfaces. The system works best on two specific workstation platforms: The Sun SPARCstation 10/512 and the Digital DEC 3000/400. On the Sun SPARCstation, VideoPix and SunVideo cards are used for video capture, and a standard X server for video output. Sun's audio hardware and software are used for audio input and output. On the Digital DEC 3000/400, DEC's J300 card and the Vidboard [9], a LAN-based video capture subsystem, are used for video capture. The standard X server is used for video display, and DEC's audio hardware and AudioFile [10] software are used for audio input and output.

Tcl is an excellent programming language for out-of-band control in the VuSystem. Its extensibility and simple interface to C is used to a great extent in the VuSystem through object commands. The simplicity of the language and its interpreted nature provide for the rapid prototyping of new VuSystem applications. Its simple type system speeds the implementation of remote-evaluation support for distributed applications. Finally, the introspective features of Tcl and the VuSystem ease the development of interactive visual media programming systems.

References

[1] D. J. Wetherall and C. J. Lindblad, "Extending Tcl for Dynamic Object-Oriented Programming," *submitted to the 1994 USENIX Symposium on Very High Level Languages*, October 1994.

[2] C. J. Lindblad, D. J. Wetherall, D. L. Tennenhouse, "The VuSystem: A Programming System for Visual Processing of Digital Video," *Proceedings of ACM Multimedia 94*, October 1994.

[3] C. J. Lindblad, D. J. Wetherall, W. F. Stasior, J. F. Adam, H. H. Houh, M. Ismert, D. R. Bacher, B. M. Phillips, D. L. Tennenhouse, "ViewStation Applications Intelligent Video Processing Over a Broadband Local Area Network," *Proceedings of the 1994 USENIX Symposium on High Speed Networking*, August 1994.

[4] C. J. Lindblad, "A Programming System for the Dynamic Manipulation of Temporally Sensitive Data," MIT/LCS/TR-637, MIT Laboratory for Computer Science, Cambridge, MA, August 1994.

[5] D. J. Wetherall, "An Interactive Programming System for Media Computation," SM Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1994.

[6] D. R. Bacher, "Content-Based Indexing of Captioned Video," SB Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1994.

[7] D. L. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Wetherall, D. Bacher, and T. Chang, "A Software-Oriented Approach to the Design of Media Processing Environments," *Proceedings of the International Conference on Multimedia Computing and Systems*, May 1994.

[8] J. K. Ousterhout, "Tcl and the Tk Toolkit," Addison Wesley, 1994.

[9] J. F. Adam, "The Vidboard: A Video Capture and Processing Peripheral for a Distributed Multimedia System," *Proceedings of the ACM Multimedia Conference*, August 1993.

[10] T. M. Levergood, A. C. Payne, J. Gettys, G. W. Treese, and L. C. Stewart, "AudioFile: A Network-Transparent System for Distributed Audio Applications," *Proceedings of the USENIX Summer Conference*, June 1993.

[11] Apple Computer Inc., "Hypercard (Version 2.2)," Apple Computer Inc., 1993.

[12] Apple Computer Inc., "Inside Macintosh: Quicktime, Inside Macintosh: Quicktime Components," Addison Wesley, 1993.

[13] Microsoft Corporation, "Microsoft Video For Windows Users Guide," 1992.

[14] ISO/IEC JTC1/SC2/W10, "Digital Compression and Coding of Continuous-Tone Still Images," IEC Draft International Standard 10918-1, 1992.

[15] ISO/IEC JTC1/SC29, "Coded Representation of Picture, Audio, and Multimedia/Hypermedia Information," Committee Draft of Standard ISO/IEC 11172, 1991.

[16] C. Williams and J. Rasure, "A Visual Language For Image Processing," *IEEE Computer Society Workshop on Visual Languages*, Skokie, Illinois, 1990.

[17] P. J. Asenta and R. R. Swick, "X Window System Toolkit: The Complete Programmer's Guide and Specification," Digital Press, 1990.

[18] C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, pp. 30-42, July 1989

TkPerl— A port of the Tk toolkit to Perl5

Malcolm Beattie

Oxford University Computing Services

mbeattie@sable.ox.ac.uk

ABSTRACT: TkPerl is a port (work in progress) of the Tk toolkit to Perl5. rr *. It takes advantage of Perl5's object oriented features and magic variables to implement the Tk toolkit in Perl5. Nothing passes through the Tcl parser so knowledge of Tcl syntax is not required to use TkPerl. TkPerl is freeware (distributed under the GNU General Public License) and is currently in alpha testing. Section 1 of the paper introduces TkPerl and is followed by a brief section on its target uses.. Since TkPerl relies heavily on the object oriented features of Perl5 (which is itself only just into beta test), section 3 explains how Perl5 implements classes, objects and methods. Section 4 discusses the differences between Tk/Tcl and TkPerl both at scripting level and at C level. Section 5 looks at some of the porting issues and problems and how Tcl conventions affect the design of Tk itself. Section 6 explains the current intentions for the future of TkPerl and section 7 gives availability information.

1. Introduction

TkPerl started life in November 1993 as an exercise in looking at the hooks Perl5 provided for interfacing to external C functions. I wanted to write a graphical front end to a network client program and for various reasons I wanted to write it in Perl. I was familiar with Tcl and Tk and took a close look at the internals of Tk. Much to my surprise, the code had few dependencies on Tcl parsing and commands (as opposed to Tcl tools such as the hash table functions). I decided to try to "port" Tk to Perl by replacing the Tcl-dependent parts of Tk with Perl equivalents. By that, I mean removing the Tcl-as-scripting-language parts of Tk (not the Tcl-as-C-toolkit parts) and replacing them with a Perl-ish interface. Most importantly,

- callbacks to Perl were to happen directly, without passing through the Tcl parser;
- Perl variables were to be traced for widgets like checkbuttons and radiobuttons so that the widgets could update themselves when required; and
- the Perl interface to the TkPerl toolkit should make use of the object oriented features of Perl5— classes, objects and methods.

A more irritating obstruction to a clean port than those mentioned above became evident as work progressed— I will discuss this later.

TkPerl is being developed under OSF/1 and Linux. It has also been tested under various versions of SunOS, Ultrix, Solaris, NeXT and a few other platforms too. It is released under the GNU General Public License. Since the first public release of TkPerl, there have been over 1000 downloads of it by anonymous ftp from its home site in the UK (as of 1 September 1994). There is also a mirror site in the US for which I do not have statistics.

The current (alpha4) release of TkPerl is in the form of an extension to Perl5. This means that it follows certain conventions for its building, loading and programmer interface. It can be built at the same time as Perl itself as a static or dynamically loadable extension or it can be built separately from Perl as a dynamically loadable extension. For dynamic loading to be possible, it must be supported by Perl on that

Tk/Tcl, written by John Ousterhout.

platform. Currently, at least the following platforms are supported for this: OSF/1, SunOS, Solaris, Linux (via GNU dld) HPUX and NeXT. If the Tk extension is to be built at the same time as Perl, the TkPerl source distribution can simply be extracted and "Tk" appended to the list of extensions that Perl's Configure asks about. The Tcl include file directory may need to be specified beforehand, though. The programmer interface for this extension is to use the command "use Tk" near the top of the script. That causes the extension to be dynamically loaded where necessary, bootstrapped, initialised and to import the necessary symbols for subroutines and variables from the Tk package namespace. The Tk extension co-exists happily with other extensions, without any namespace pollution other than the documented class names and public routines such as **tkpack** or **addasyncio.**

2. Target uses for TkPerl

Rather than letting comparison between Tk+Tcl and Tk+Perl5 become a comparison between Tcl and Perl5, I'd like to suggest a somewhat different viewpoint for TkPerl. Many large applications are written in Perl and rely on Perl's native features (regular expressions, associative arrays, low-level system access) for power, Perl's compilation and optimisation for speed, Perl's warnings and perldb.pl for debugging and Perl's taint checking for security. Looked at in this way, Perl is comparable to C and TkPerl should be compared to the combination Tk/C rather than the combination Tk/Tcl. There is, as yet, no documentation on the C interface to Tk although I believe interest in this has been increasing recently. For example, TkPerl provides the asynchronous I/O facilities of Tk: these are available with the C interface to Tk but not the Tcl interface. TkPerl provides access to the Tcl interpreter used by Tk via a **tclcmd**() routine, although the widget command procedures are not available in there. A TkPerl script can use the Tcl interpreter in the same ways as a C program can: for allowing flexible user configuration and for allowing access to user-written C functions. I think that the latter is less likely to be relevant in TkPerl, though, since Perl5 provides similar functionality.

Having made the above suggestion, I should probably point out that, in practice, many potential users of TkPerl have seen it in a different light. I have been in touch with many perl users who are eager for access to the Tk toolkit but who do not wish to use Tcl for one reason or another.

3. A brief introduction to Perl5 objects and methods

Since TkPerl makes much use of the object oriented features of Perl5 and Perl5 itself is still only in beta test, I will explain a little bit about how classes, objects and methods are implemented in Perl5.

3.1. Packages

Those familiar with Perl4 will know that all subroutine names and variable names are qualified (even if only implicitly) by a package name. Variables can also be explicitly qualified by a package. The syntax for scalar variable **bar** in package Foo is $Foo'bar in Perl4 and $Foo::bar in Perl5. Variables which are not qualified by an explicit package are by default put in package "main". That default package can be changed by a command such as

```
    package Foo;
```

which affects all variables from there to the end of the (lexically scoped) containing block.

3.2. References

In Perl4 variables are either scalars (strings or numbers), arrays (indexed by integers) or associative arrays (indexed by strings). Those fluent in Perl will realise that I haven't mentioned globs or filehandles but we will not concern ourselves with those here. In Perl5, there is another scalar type too. That type is a "reference" which can hold a reference to any other type (scalar, array or associative array). Syntactically, one prepends a backslash to any other variable to get a reference to that variable. Recall that we prefix identifiers with "$" for a scalar variable, "@" for an array or "%" for an associative array. Array constants can be comma-separated lists surrounded by parentheses. Associative array constants are of the same form but are taken in pairs of key followed by value.

Given the code

```
@evens = (2, 4, 6, 8);
$evenref = \@evens;
```

the scalar variable $evenref is a reference to the array @evens (and $evenref knows internally what type of variable it references). The syntactical rule in Perl5 for dereferencing references is as follows. Wherever it is legal to have a sequence of letters identifying a variable, it is also legal to have a reference to a variable of that type. For example, given the above code, @$evenref is the actual array @evens. The 0th element of the array can be referred to as either $evens[0] or $$evenref[0].

In Perl4, subroutines are defined with the syntax

```
sub foo {
    ...
}
```

and invoked with the syntax

```
&foo($arg1, $arg2, $arg3);
```

In Perl5 syntax, the "&" can be dropped when invoking subroutine foo and scalar variables can hold a reference to a subroutine. The code snippets

```
$fooref = \&foo;
&$fooref($arg1, $arg2, $arg3);
```

and

```
foo($arg1, $arg2, $arg3);
```

both invoke subroutine foo in the same way as the preceding example.

3.3. Blessing

Access to the object oriented features of Perl5 is by means of the simple but powerful **bless** command. This command takes as argument a scalar variable which contains a reference and "tags" that variable which the name of the current package (or with the package named by the optional second argument of the **bless** command). By "tags", I mean that Perl remembers that that package is associated with the variable. By slight abuse of language, the reference is called a *blessed reference*. No variable's value is affected, but the **ref** function applied to that reference will return the name of the package into which it is blessed (or the special out-of-band value **undef** if it is not blessed at all).

3.4. Classes and methods

How does the **bless** command make classes and methods possible? Well, suppose $obj is a scalar variable holding a reference blessed into package Foo. Leaving aside inheritance for a moment, Perl5 uses the syntax

```
$obj->somemeth($arg1, $arg2);
```

to mean the same as

```
&Foo::somemeth($obj, $arg1, $arg2);
```

In other words, Perl looks for subroutine &somemeth in package Foo and invokes it with $obj prepended to the other arguments passed. What if package Foo has no subroutine &somemeth? That is where inheritance comes in. The special array variable @ISA in package Foo is taken to be an (ordered) list of package names. If there is no subroutine &somemeth in package Foo, then Perl5 searches through all the packages named in @Foo::ISA for a subroutine &somemeth and executes the first one it comes to. If Perl *still* cannot find a subroutine &somemeth, then there are other fallbacks which allow autoloading and universal classes but I will not go into those here.

If you apply the -> operator with a bare word in place of $obj above, then it is taken to be a package name in which to start the subroutine search. The first argument passed to the subroutine it finds is that bare word. For example, assuming a subroutine &Foo::new exists, the code,

```
$foo = Foo->new($arg1, $arg2);
```

calls subroutine &new in package Foo with arguments ("Foo", $arg1, $arg2). The example above is rather a special one because, purely by convention, the subroutine &new in a package is the constructor for that package. In other words, it is expected to return a scalar variable containing a reference blessed into that package. The resulting scalar (assigned to $foo above) can be used in method calls such as

```
$foo->flash();
$foo->disable();
@textconf = $foo->configure("-text");
```

assuming that package Foo has defined appropriate subroutines &flash, &disable and &configure. A *class* in Perl5 is (by convention) just a package which has a constructor and whose subroutines are intended to be called as methods. An *object* in Perl5 is (again, by convention) just a blessed reference returned by a class. When a constructor for a class wants to return an object, it

- creates a reference to a variable (usually an array or an associative array) which contains any public or private data for that object;
- blesses the reference; and
- returns the reference.

When any of its subroutines (methods) are called, the subroutine dereferences its first argument to get at the object's data.

3.5. Syntactic sugar

There are some syntactic variations which are useful for common idioms. Method calls of the form

```
$obj->meth($some, $args);
$foo = Class->constr($some, $more, $args);
```

can also be written in the form

```
meth $obj ($some, $args);
$foo = constr Class ($some, $more, $args);
```

There is also syntax (which is rather more than mere sugar) which enables creation of references to anonymous arrays, anonymous associative arrays and anonymous subroutines. For example, the code

```
$listref = [1, 2, 3, [7, 8, 9]];
$hashref = { "key" => "value", "anotherkey" => "anothervalue" };
$callback = sub { print "Hello world\n" };
```

creates three references. Dereferencing them, we get

- @$listref, which is an array with four elements. The fourth element is a reference to an array with the three elements (7, 8, 9).
- %$hashref, which is an associative array with keys "key" and "anotherkey" whose respective values are "value" and "anothervalue". The symbol => is syntactic sugar for a plain comma but may be used in the future for something a bit cleverer.
- &$callback, a subroutine which prints a traditional greeting when invoked.

4. Differences between Tk/Tcl and TkPerl

This section outlines the differences between Tk/Tcl and TkPerl. The section splits fairly naturally into two parts, according to the user of TkPerl. There is, of course, no need to mention the end-user of a given Tk program in the comparison since all differences are at a lower level. I would think that the majority of TkPerl users would give their attention to the differences between TkPerl and Tk/Tcl as regards writing X clients purely at the script level. Any differences between additional non-X C functions that are bound in really fall outside the bounds of Tk-specific paper. However, writers of new widgets may well be interested in the differences between TkPerl widget code and Tk/Tcl widget code. For a simple widget, there is likely to be very little difference. Moreover, if any widget is written from scratch it can be written in such a way that minimal changes are necessary to go from a Tk/Tcl widget to a TkPerl widget.

4.1. Differences at scripting level

Provided the Tk extension is available to the perl executable (either statically linked into the perl executable or, more likely, dynamically loadable from a sharable object file), any perl script just needs the command

```
use Tk;
```

somewhere near its beginning to be able to make use of TkPerl. An average script using TkPerl would then do some initial processing followed by

```
$top = tkinit();
```

to initialise Tk and prepare a top level widget. Then it would create lots of widgets and finally call **tkmainloop** which invokes the main event loop. Callback are done to subroutines (possibly in their guise as methods), which can be defined anywhere, as usual in Perl. The **tkinit** command takes up to three arguments (all optional) to allow non-default application name, DISPLAY and X server synchronisation (this last for debugging).

4.1.1. Creating widgets

Whereas widgets in Tk/Tcl are created by a Tcl command whose name is that of the widget class (e.g. **button**), in TkPerl each widget class is a Perl class (e.g. Button), with a constructor method **new**. The **new** method takes as first argument either the parent widget object or else the Tk pathname you want to give the widget. (The return value of **tkinit** is a widget object referring to a toplevel widget. Tk/Tcl sets this widget up automatically at script start up and calls it "."). Later arguments are optional and are pairs of initial configuration requests.

Example

```
Tk/Tcl      button .b -text foo
TkPerl      $b = Button::new($top, "-text" => "foo");
```

$b above is a reference to the pathname, blessed into the appropriate widget class.

4.1.2. Calling widget methods

In Tk/Tcl creation of a widget causes creation of a new Tcl command whose name is the widget's pathname. Invoking this command with first argument **foo** invokes method **foo** on that widget. In TkPerl, the value returned from the constructor **new** of a widget class is a blessed reference. Widget methods are real methods in that class and so one can use the blessed reference to invoke them.

Example

```
Tk/Tcl      .b flash
            .b configure -textvariable bar
TkPerl      $b->flash();
            $b->configure("-textvariable" => $bar);
```

There are a few minor differences in names: delete, index and select are Perl keywords and have been renamed tkdelete, tkindex and tkselect. Tcl "submethods" such as

```
.some.widget scan mark
```

have been turned into single methods: scanmark, scandragto, selectclear, selectadjust, selectfrom, selectto. When a **configure** method is used to return information about a widget option TkPerl returns a standard Perl list with the same information returned by Tk/Tcl. When the **configure** method is used to return information about all options of a widget TkPerl returns a list each of whose elements is a reference to an information list about one particular option. For example, after executing

```
@conf = $w->configure();
```

$conf[0] is a reference to a list containing either five elements (for an ordinary option) or two elements (for an alias option).

4.1.3. Callbacks

Configuration options where Tk/Tcl has a callback command (e.g. **-command, -yscrollcommand**) use slaves and methods instead in TkPerl (**-slave**, **-method**, **-yscrollslave**, **-yscrollmethod**). A slave/method pair is intended to be a flexible way to define a callback which is either

- an ordinary subroutine together with a piece of clientdata; or
- an object together with a method to invoke on it.

Suppose you have a widget $w which you configure as follows:

```
$w->configure("-slave" => $slave, "-method" => $method);
```

Callback behaviour depends on whether $slave is a blessed reference or not. If $slave is a blessed reference then the method $method is invoked on the object $slave. If $method is "foo" then it is the same as doing

```
$slave->foo();
```

in pure Perl. If $method is blank then no callback happens. On the other hand, if $slave is not a reference (in particular, if $slave is not defined), then $method is taken to be an ordinary subroutine and is called with $slave as its first argument (or **undef** if $slave is not defined). $method itself can be either an ordinary variable which is taken to be the name of a subroutine or else it can be a subroutine reference. The subroutine reference can either be of the form \&foo or an anonymous subroutine sub { ... }. This latter form gives the TkPerl programmer a way of writing callback code right next to the action which should trigger it. Moreover, the callback code is still compiled by Perl just as any ordinary subroutine is. This gives better performance than **eval** or its equivalent, especially for callbacks such as mouse dragging callbacks triggered for motion events. Even if the name of a subroutine is given as a $method argument, the lookup from subroutine name to internal Perl code pointer happens only the first time the callback is triggered. Thereafter a cached code pointer is used, again making for better performance. In all cases, the callback routines are passed extra arguments in precisely the same situations as in Tk/Tcl (e.g. scrollbar callbacks).

4.1.4. Access to event fields

In Tk/Tcl event callbacks are specified as Tcl command strings in which magic cookies beginning with the percent character "%" are replaced with data from fields from the event associated with the callback. In TkPerl, no magic cookies are involved but the magic variables $Tk::EvW, $Tk::EvA and so on all refer dynamically to the appropriate data. One slight addition: since $Tk::EvW returns a widget pathname and sometimes blessed references are more useful, the subroutine &EvWref returns a blessed reference to the appropriate widget.

4.1.5. Variable tracking

In configuration options such as **-variable** for radiobuttons where Tk/Tcl expects the name of a Tcl variable (as a string), TkPerl expects a genuine perl variable.

Example

```
Tk/Tcl     .r configure -variable foo -value green
TkPerl     $r->configure("-variable" => $foo, "-value" => "green");
```

Any alteration to that variable is tracked by the widget and it updates itself as necessary.

4.1.6. Binding events

Instead of Tk/Tcl **bind**, TkPerl has a **tkbind** command, since **bind** is already a Perl keyword:

```
tkbind($w, "<SomeEventName>", \&someaction);
```

$w can be a blessed reference to a widget or a widget pathname or a string corresponding to a widget class (e.g. "Button") or "all". The third argument is the method or subroutine half of a slave/method pair, as described in a previous section. An optional fourth argument is the corresponding slave/clientdata.

4.1.7. Processing a filehandle asynchronously

There is no Tk/Tcl equivalent for this, although Tk offers the service as C routine. The TkPerl command

```
    addasyncio(FILEHANDLE, $selectfor, $method);
```

(with an optional fourth argument, $slave) arranges for the method to be called whenever one of the "interesting" conditions named in the $selectfor argument happens on FILEHANDLE. An "interesting" condition here means one where the file descriptor underlying the Perl filehandle FILEHANDLE becomes readable, writable or has an exception occur on it. When $method is called, it is called with arguments that tell it which condition(s) actually happened. When the conditions include being readable, one of the arguments is the number of bytes available to be read with blocking (if the O/S makes this information available).

4.1.8. Widget objects and inheritance

As alluded to above in a number of places, there are different ways of referring to widgets. You can use Tk pathnames, but you lose the advantages of perl objects and this use may be deprecated in a future release. The return value of each widget constructor method **new** is a reference to a string (the new pathname, in fact), blessed into the widget's class. When the first argument to a widget **new** method is a widget object (i.e. not a direct string pathname), an automatically generated unique pathname of a child of that widget is used instead (cf. the Xt toolkit and the Athena widget set). One way of constructing widget objects not yet referred to is very useful for simplifying widget inheritance for user-written composite widgets. A (blessed) reference to an associative array is acceptable as a widget object. In this case, when a widget method is applied to it (e.g. via @ISA inheritance) it looks for a key of that widget's classname and takes the corresponding value. If the key is not present it looks for the key "Default." The value found is then decoded as a widget by trying all the previously mentioned interpretations. For example, given code of the form

```
    package Scrolledlistbox;
    @ISA = (Listbox, Scrollbar);
    sub new {
        # create a new listbox $l
        # create a new scrollbar $s
        # link the two together
        bless {Listbox => $l, Scrollbar => $s};
    }
```

and the assignment

```
    $sl = Scrolledlistbox::new($parent);
```

in a TkPerl script, $sl will understand the methods of both the Listbox class and the Scrollbar class. Those methods will be applied to the appropriate widget ($l or $s). Certain TkPerl commands look for other special keys (e.g. **focus** looks for key Focus). Since the widget object is a reference to an associative array, any public or private data fields for the object can be held as values in that associative array.

4.2. Differences at widget-writing level

We have already seen how composite widgets can be implemented in Perl scripts by imaginative use of inheritance. This section concerns how widgets themselves are implemented in C. This is more closely related to the following section on porting issues than to the preceding section on Perl classes. The C programmer's interface for writing new widgets is still rather fluid, especially as regards constructors. A future version of TkPerl will probably have a base Widget class from which fundamental methods can be inherited. In particular, the **new** constructor and the **configure** method for most widgets could be inherited. The widget writer would just register a class record containing such things as

- the classname,
- the configspecs table, and
- a list of method names.

The methods themselves can be written in perl or in C. If they are to be written in C, the methods can be in the form of XS files, which make use of the xsubpp pre-processor distributed with Perl5.

4.2.1. Widget constructors and methods in C

The widgets written in C which are currently distributed with TkPerl are all ports of the original Tk/Tcl widgets. They do not make use of Perl's XS pre-processing format because they were ported before xsubpp was developed. When the command "use Tk" is issued in a script, the sharable object code for Tk is loaded (if not already statically linked into the perl executable) and Tk's bootstrap function is invoked. This calls an initialisation routine for each widget which passes its class structure to TkPerl's **init_widget** utility. The fields in the class structure include the classname, a function pointer for the widget's constructor, a function pointer for the widget's methods and an array of integers corresponding to the widget's methods. The constructor function for widget class Foo is a slight modification of what Tk/C would call **Tk_FooCmd** and what a Tk/Tcl programmer would create with

```
foo .some.widget.path
```

Each element of the array of integers is an enumeration constant which indexes into a global array of standard method names. The **init_widget** function binds the constructor function to the Perl routine **&new** in package Foo. For each entry in the array of method integers, it binds the corresponding method name to the function pointer for the widget's methods. When the function is invoked, it will have available the method index integer for the name which invoked it. That function pointer is to a modification of what Tk/Tcl calls the **Tk_FooWidgetCmd** function and what the Tk/Tcl programmer would call as the ".some.widget.path" command from the example above. In TkPerl this function consists of a large **switch** statement which implements each method as a **case** instead of parsing a string corresponding to the method name, as Tk/Tcl does.

TkPerl offers three extra configuration option types to implement callbacks and variable tracking. In Tk/Tcl, callbacks are strings interpreted as Tcl commands and tracked variables are strings interpreted as names of Tcl variables. In TkPerl, callback subroutines are compiled and variable symbol lookup happens at compile time so new configuration types are necessary. Instead of a TK_CONFIG_STRING entry in the configspecs table, tracked variables use TKP_CONFIG_VARIABLE, methods use TKP_CONFIG_METHOD and slaves use TKP_CONFIG_SLAVE. In the widget structure itself, a tracked variable is stored with C type **SV *** and a method/slave pair is stored in a **struct tkpcallback** which automatically handles caching of method lookups for optimising speed. Invoking a callback from application code is done by pushing any extra arguments onto the Perl stack (if necessary) and invoking **TkpCallback(callback, n).** Here, **callback** is the relevant **struct tkpcallback** and **n** is the number of extra arguments to pass to the callback (over and above the slave/clientdata argument that is automatically passed). The possible conversion of strings to subroutine references and the caching of Perl's internal pointers to them is all handled transparently by TkpCallback.

4.2.2. Utility functions

TkPerl supplies utility functions in C for the widget writer in addition to TkpCallback and the configspecs types mentioned above. These cover such things as

- converting from an SV (the internal type for Perl variables) to a widget record or Tk window structure;
- setting up traces on Perl variables (the equivalent of Tcl_TraceVar); and
- generating a new widget given a pathname or parent widget object.

5. Porting issues

At first sight, there were only two areas which would require significant work before a minimal Perl version of Tk would run. Those were:

(1) Many Tk/Tcl widgets have resources such as **-command** to store text for callbacks which are later interpreted as Tcl command strings using **Tcl_Eval()**.

(2) Many Tk/Tcl widgets use the variable tracing facilities of Tcl (such as **Tcl_TraceVar**()) to carry out a C function whenever a Tcl variable is written to. For example, a checkbutton widget traces the Tcl variable named by its **-variable** option and sets or resets its visual marker accordingly whenever that variable is modified.

Perl5 provides similar facilities for C programmers but with rather different interfaces. On top of this, the infrastructure of each Tk widget command procedure (the C function bound to a widget's pathname) would need to be rewritten. This would just involve changing a series of "if-then-else" string comparisons to a **switch** statement to handle the way Perl5 calls XSUBs (C functions bound to perl subroutines, called usersubs in Perl4).

There turned out to be a more serious problem with the Tk/Tcl code as far as porting it to Perl5 was concerned. This wasn't a localised problem but a more general one: the argument passing conventions used by Tk/Tcl. Not only the Tcl command procedures but also most of the internal Tk widget procedures used argc/argv style arguments. In other words the callers would convert any arguments in to an array of null terminated strings and the called function would parse out the appropriate data types again. This is highly suited to interfacing to Tcl since the interface between Tcl scripts and their corresponding C functions uses this model.

For Perl5, however, it is far from the best way of handling argument passing. Perl5 shields the programmer both at scripting level and at the C level from data-type problems. A Perl5 XSUB receives its arguments on a global stack, which can be treated just like an array of opaque pointers. Moreover, by writing source using the new XS language and its xsubpp pre-processor, one can avoid dealing with the stack or data-type conversions altogether. One need only name the C types of the variables one wants to be passed and the xsubpp pre-processor generates C code to handle the internal Perl stack and the necessary data conversions. All Perl variables are internally modelled as *scalar values* (SV's), a generic opaque structure which knows what sort of data it contains. There are documented macros for getting at that data and the Perl5 internals handle data-type issues automatically. Not only can they give you the contents of that variable as whatever data-type you want (for example string, integer or double) but any data-type conversion happens only where necessary. That is because an SV can hold the data in multiple formats at the same time. Furthermore, Perl5 supports using the SV as an lvalue and supports data-types such as lists, associative arrays and references to any data-type at all.

Carrying over the argument passing conventions of Tk/Tcl into TkPerl would have made for clumsy code. For example, suppose a numeric argument is passed to a Perl XSUB. (Recall that an XSUB is a perl subroutine that happens to call out to an external function). The Perl compiler will already have chosen its appropriate type (integer or double) when the script was parsed. The XSUB has immediate access to a C variable of type int or double. For a direct interface to the C functions which implement the Tk internals, that int or double would need converting to its string representation. Then it would be passed to a function whose first job would be to convert its string argument back into a numeric. This is unavoidable for C functions directly called by the Tcl parser because every piece of data in a Tcl script is a string and all arguments are passed into C functions as string arrays. However, it seems a crying shame to take this route in Perl, where all necessary conversions, optimisations and caching of values has already been done. It also seems rather a shame that there is not a stricter boundary between the two apparent "layers" of Tk/Tcl. That is,

- the upper layer consisting of C functions bound into the Tcl interpreter and called with argc/argv pairs, and
- the lower layer of functions which deal directly with widget fields and Tk's windows.

In particular, consider the C function bound to a typical widget pathname command in Tk/Tcl. It not only parses its argument strings (for example **get, invoke,** or **nearest**) but it also carries out most of the resulting "method" commands. Those auxiliary functions which it does make use of tend to be declared **static,** to accept strings as arguments and do their own parsing. It is not possible to write a separate TkPerl function which takes values of the required type and calls directly to the low level functions. One workaround is to copy the whole source file and modify the low level functions slightly to accept arguments of the correct type without parsing them from strings. Another workaround is to convert the arguments that Perl passes back into strings, dynamically create a new array of strings and then call into the low level routine. The latter workaround still needs the whole source file to be copied since the low level functions are declared **static** and so cannot be linked in from the standard Tk library statically, let alone dynamically.

6. Future plans

TkPerl is still alpha-code. It lacks

- one of the standard Tk widgets (the Text widget);
- some of the standard bindings (such as menu bars and accelerators);
- Perl-ish versions of some auxiliary commands; and
- the Tk/Tcl **send** command (which I intend to implement with a more general Perl implementation of remote proxy objects).

  Perl5 itself is still in beta-test. Although the programmer interface to its internal functions aren't changing as frequently as they used to do, updating TkPerl to make use of newly documented interfaces and conventions is still non-trivial.

In the short term, I will continue tracking releases of Perl5, stabilise the current set of widgets and complete the set supplied, provide more Perl-ish interfaces to the Tk support commands and complete Tk.pm, the TkPerl equivalent of tk.tcl, written in Perl rather than Tcl. In the short to medium term, a decision has to be made. Currently, TkPerl is fairly close in many ways to Tk/Tcl. Apart from having widgets as Perl5 classes and their methods as true methods (allowing widget inheritance) there are few other major differences. It would be possible to take further advantage of Perl's object oriented features and rewrite major parts of Tk to behave differently. For example,

(1) the event delivery system could be rewritten to propagate events using Perl5's method inheritance,

(2) the widget configuration system could be rewritten to use method chaining (like with Xt's **set_values** method) simplifying widget inheritance even more,

(3) the main widget procedures could be allowed to be real Perl methods. Ways of interfacing Perl5 directly to C libraries are being developed and one could imagine new widget code being written directly in perl.

(4) Perl's **tie** command opens up many possibilities for useful interfaces to widgets.

However, it could also be argued that any major divergence of TkPerl from Tk/Tcl may cause more harm than good. It would make folding in future changes to Tk/Tcl harder, or even impossible, and would mean that programmers would be less able to cross to and fro between writing Tk/Tcl code and TkPerl code. Should the goal of TkPerl be to provide non-Tcl speakers with an alternative way of using the popular Tk toolkit or should the goal be to provide a native Perl GUI based on Tk? My current feeling is that the latter goal is more worthwhile.

7. Availability

TkPerl is freeware distributed under the GNU General Public License. It is currently available for anonymous ftp from ftp.ox.ac.uk in directory `/src/ALPHA`. The source-only distribution of the current alpha release is available as file `tkperl5a4.tar.gz`, a tar archive compressed with GNU gzip. Distributions which include executables (perl executables with the Tk extension linked in or else sharable object files for dynamic loading) may also be made available for some platforms. The most appropriate forum for discussion of TkPerl is probably the newsgroup `comp.lang.perl`. I am happy to receive email about TkPerl but can provide no guaranteed support for it.

Although TkPerl has been developed in my own time, I have been doing much of the development on the general Unix service system of my employers, Oxford University Computing Services. I would like to thank them for their encouragement and for an attitude towards free software that makes this kind of development work possible.

Malcolm Beattie gained his BA in Mathematics at Oxford University in 1989 and continued there to work towards his DPhil in Algebraic Topology. He joined Oxford University Computing Services in 1992 as a systems programmer and gained his DPhil in 1993.

Rapid Programming with Graph Rewrite Rules

Andy Schürr

Lehrstuhl für Informatik III, RWTH Aachen,
Ahornstr. 55, D-52074 Aachen, Germany
e-mail: `andy@i3.informatik.rwth-aachen.de`

Abstract: Graphs play an important role within many areas of computer science and rule-based languages are more and more used to describe complex transformation or inference processes. Nevertheless, their combination in the form of graph rewriting systems were more or less unknown among computer scientists for a long time. Nowadays, the situation is gradually improving with the appearance of a number of graph rewriting system implementations. Currently, the multi-paradigm language PROGRES is the most expressive implemented graph rewriting language. It has the flavor of a visual database definition and programming language and combines the advantages of attributed (graph) grammars with rule-oriented as well as imperative programming. An integrated set of tools supports editing, analyzing, and debugging of applications as well as translation into procedural programming languages (C, Modula-2).

1. Introduction

Graphs play an important role within many areas of computer science and there exists an abundance of visual languages and environments which have graphs as their underlying data model [12, 35]. Furthermore, rule-based rewriting languages are well-suited for the description of complex transformation or inference processes on complex data structures. Although graphs and rule-base rewriting systems are quite popular among computer scientists, their combination in the form of *graph rewriting systems* were more or less unknown for a long time. Nowadays the situation is gradually improving with the appearance of graph rewriting system implementations like AGG [19], GraphED [15], PAGG [11], or GOOD [14]. Unfortunately, all these systems have their main focus on the definition of graph rewrite rules only, and they either have no proper means to model integrity constraints and derived properties of data structures or they do not offer any constructs to combine basic rewrite rules to more complex transformation processes.

The language PROGRES is the first attempt to overcome these deficiencies of graph rewriting systems. It is a *strongly typed multi-paradigm language* with well-defined syntax, static and dynamic semantics. Its name is an acronym for *PRO*grammed *G*raph *RE*writing *S*ystems which are the language's underlying formalism and are in turn defined by means of nonmonotonic logic and fixpoint theory [26, 27]. Being a mixed textual and diagrammatic language, it permits quite different styles of programming and supports

- graphical as well as textual definition of graph database schemas,
- declaration of derived graph properties in the form of derived node attributes, node sets, and binary relations between nodes,
- rule-oriented and diagrammatic specification of atomic graph rewriting steps by means of parametrized graph rewrite rules (productions) with complex preconditions, and
- imperative programming of composite graph transformation processes by means of deterministic and nondeterministic control structures.

In some respect, PROGRES is rather similar to so-called "visual database languages" as for instance GraphLog [6] or QBD* [2]. But note that these languages are more or less "pure" query languages which do not support the definition of complex transformation processes. In contrast, PROGRES and its *programming environment* support definition and manipulation of data. They are already in use for

- specifying tools and data structures of software engineering environments [8, 22],
- describing tools for process modeling and configuration management in CIM environments [28],
- and defining the semantics of a visual database query language [1].

This paper provides an overview of the language PROGRES and its programming environment. Section 2 introduces red/black trees as a running example and shows how graph rewrite rules simplify their implementation considerably. Section 3 afterwards describes the PROGRES programming environment and explains the main ideas of its underlying execution machinery. Section 4 offers a short comparison with related work and section 5 the paper's conclusion.

2. The Language PROGRES

The language PROGRES was mainly designed as a specification language for syntax-directed tools of software engineering environments [8, 22]. But gradually it became obvious that it is also a kind of visual database programming language, and that it might even be used as a very high level language for implementing graph-like abstract data types. The running example, red/black trees, has been selected in order to demonstrate the advantages of *programming with graph rewrite rules* in contrast to the usage of conventional programming languages. It has rather complex tree restructuring operations, but its data model is just a binary tree with colored nodes. Therefore, we have to emphasize that PROGRES allows for the definition of arbitrary directed (attributed) graphs, and is not restricted to the case of attributed trees or directed acyclic graphs (as for instance [13, 17]).

Directed, attributed graphs consist of typed nodes with attributes and directed typed edges between pairs of nodes. Each edge may be traversed in both directions (from source to target and vice-versa), i.e. is equivalent to a pair of pointers of Pascal-like programming languages. Furthermore, referential integrity is guaranteed, i.e. when a node is deleted, all adjacent edges are deleted, as well. Node attributes and binary relations are either extensionally defined and manipulated by graph rewrite rules (as intrinsic attributes and edges) or they are intensionally defined and maintained by the PROGRES runtime system (as derived attributes and materialized path expressions). A lazy and incrementally working evaluation process keeps derived data in a consistent state. It is a refined version of the algorithm in the graph-oriented database system Cactis [16] and part of our own database system GRAS [18]. As a consequence, PROGRES programmers do not have to worry about inconsistent pairs of pointers, dangling references, and inconsistent intensionally definable data.

Returning to our running example of red/black trees, we have to mention that we will not use colored nodes as in [4], but colored edges. This makes our specification more readable and simplifies the assignment of new colors within tree restructuring operations. Therefore, our invariances of red/black trees are defined as follows:

```
(1)  node class RBNODE
       intrinsic
         key Value : integer;
       derived
         Height : integer = max ( self.lHeight, self.rHeight );
         lHeight = [ self.-bl->.Height + 1 | self.-rl->.Height | 0 ];
         rHeight = [ self.-br->.Height + 1 | self.-rr->.Height | 0 ];
     end;
(2)  edge type bl : RBNODE [0:1] -> RBNODE [0:1];
(3)  edge type br : RBNODE [0:1] -> RBNODE [0:1];
(4)  edge type rl : RBNODE [0:1] -> RBNODE [0:1];
(5)  edge type rr : RBNODE [0:1] -> RBNODE [0:1];
(6)  path left : RBNODE [0:1] -> RBNODE [0:1] =
       [ -bl-> | -rl-> ]
     end;
(7)  path right : RBNODE [0:1] -> RBNODE [0:1] =
       [ -br-> | -rr-> ]
     end;
(8)  path parent : RBNODE [0:1] -> RBNODE [0:1] =
       [ <=left= | <=right= ]
     end;
(9)  restriction balanced : RBNODE =
       valid (self.lHeight = self.rHeight)
     end;
(10) restriction correctColoured : RBNODE =
       not with (<-rl- or <-rr-) or not with (-rl-> or -rr->))
     end;
(11) node type RBNode : RBNODE end;
(12)        ...
```

Fig. 1: Graph schema of red/black trees.

- Different nodes in a tree have different integer values as key attributes (cf. declaration (1) of fig 5).
- A red/black tree contains only one type of nodes, but different types of edges (cf. decl. (11)).
- Any parent/child edge is either red or black and leads either to a left or a right child (decl. (2)-(5)).
- Any node has at most one left and right child and at most one parent node (decl. (6)-(8)).
- The height of a subtree is the maximum number of black edges on any path to one of its leafs (decl. (1)).
- The height of the left subtree of a node is always equal to the height of its right subtree (decl. (9)).
- The target of a red colored edge is never the source of another red colored edge (decl. (10)).

As already indicated, figure 5 contains the textual PROGRES declarations of all the above mentioned properties[1]. The first declaration introduces the *abstract node class* RBNODE with an intrinsic key attribute Value and three derived attributes. Value is defined as a key attribute in order to ensure its uniqueness for debugging purposes only, since it is the task of red/black trees themselves to ensure uniqueness of these attributes and to access their nodes efficiently. The first derived attribute Height states that the height of a subtree is the maximum of its l(eft)Height and its r(ight)Height. And the l(eft)/r(ight)Height of a subtree root is

- the Height of its left/right child, if this child is reachable via a r(ed)l(eft) or a r(ed)r(ight) edge, or
- the Height of its left/right child plus one, if this child is the target of a b(lack)l(eft) or a b(lack)r(ight) edge, or
- zero, if a left/right child does not exist.

Note that the execution of a expression like "[a | b | ...]" proceeds as follows: It starts with an attempt to evaluate the subexpression a and to return its value as the result of the whole expression. If this attempt fails, then the execution proceeds with the evaluation of "[b | ...]" and so forth.

The next four lines of figure 5 declare the already used *edge types* b(lack)l(eft), b(lack)r(ight) etc. including the accompanying cardinality constraints. Any RBNODE has for instance at most one incoming and one outgoing bl edge. The following lines declare a number of derived relations by means of so-called path expressions. They introduce merely new names for graph traversals, which will be used in the sequel, and improve thereby the readability of our specification. A node y will be called the left (child) of another node x, if and only if x is the source of a bl or a rl edge with target y. A node y will be called the right (child) of another node x, if and only if x is the source of a br or an rr edge with target y. Finally, a node x is called parent of a node y, if and only if we reach x from y by following either the path left or right in reverse direction.

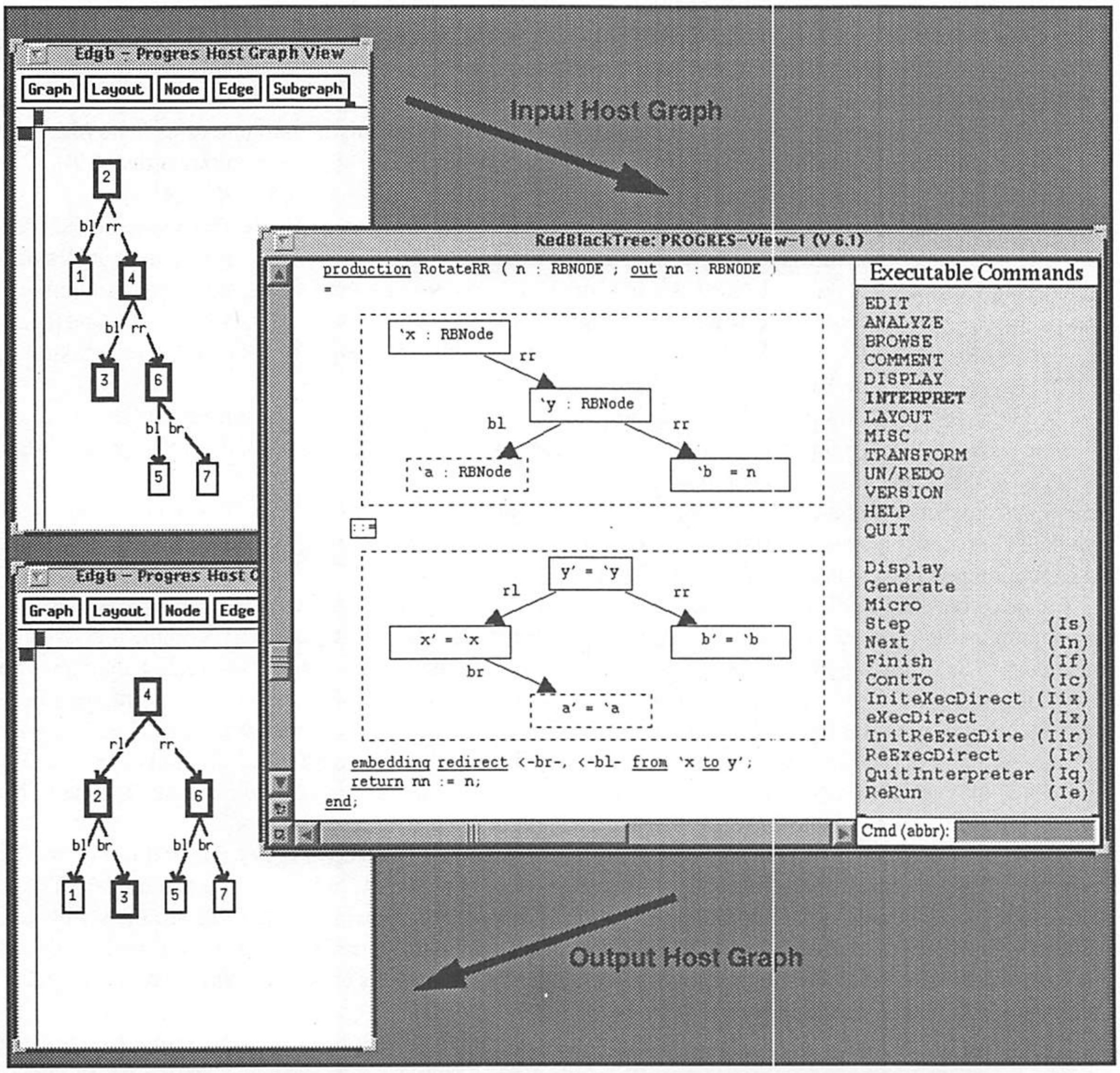
The following two *restrictions* define the more complex invariances (integrity constraints) of red/black trees. The first one requires equal heights of subtrees. The second one excludes the case that the target of a red edge is the source of another red edge. Such a restriction is an intentionally defined set of nodes with certain properties. It may either be used to select nodes within a given set of nodes, or to retrieve all nodes of a whole graph with the defined property. In the latter case, the derived node set is explicitly stored within the graph database and incrementally maintained in a consistent state.

The last declaration of figure 5 needs a more detailed explanation. It defines a *concrete node type*, which belongs to the abstract node class RBNODE. PROGRES has a stratified type system, where types of nodes are "first class objects" and may be used as parameter, variable, or attribute values. Node classes are types of node types and determine all common properties of their node type instances. Any node of type RBNode of class RBNODE has for instance the above mentioned attributes and is a legal source or target for the introduced edge types and paths. The main reason for calling types of node types "abstract node classes" instead of "meta types" was that they are members of a multiple inheritance hierarchy. Subclasses inherit the properties of their superclasses and refine them appropriately. Unfortunately, red/black trees (with colored edges) contain only one type of node and, therefore, inheritance is not needed throughout the whole paper.

```
procedure RotateRR(n) =
    y := parent[n]; x := parent[y];
    right[x] := left[y]; rightColour[x] := black;
    if left[y] # nil then parent[left[y]] := x;
    parent[y] := parent[x];
    if x = left[parent[x]]
        then left[parent[x]] := y
        else right[parent[x]] := y;
    left[y] := x; leftColour[y] := red;
    parent[x] := y;
end;
```

Fig. 2: Pseudocode for operation RotateRR.

1) The graphical representation of these declarations is a kind of extended ER-dialect; cf. right/bottom window of fig. 3.



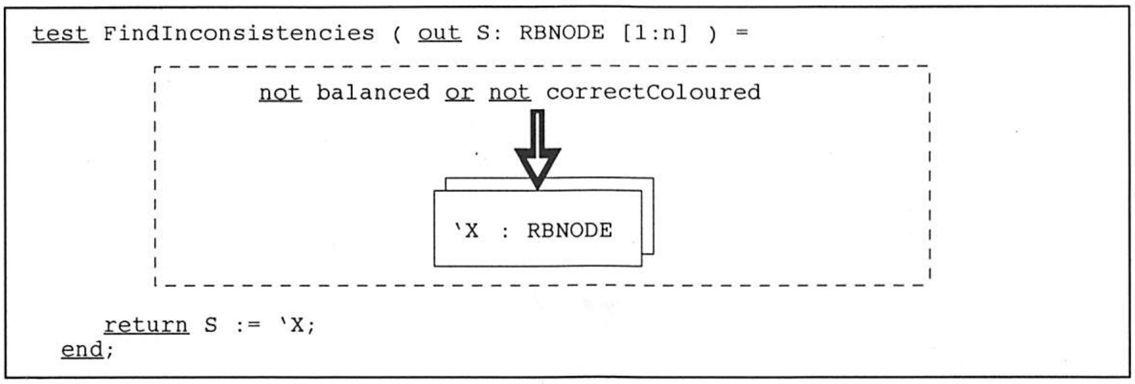
```
production RotateRR ( n : RBNODE ; out nn : RBNODE )
=
embedding redirect <-br-, <-bl- from `x to y';
return nn := n;
end;
```

Fig. 3: PROGRES definition and application of production RotateRR.

To ensure all the above mentioned properties, red/black trees must be reorganized after insertion or deletion of certain nodes. The "backbone" for the reorganization process are four subtree rotating operations and an edge recoloring operation, called Split. Figure 2 presents the slightly modified pseudocode for one of the tree rotating operations. It has the name RotateRR, since it deals with two r(ight)r(ed) edges which meet each other. It is a slightly modified version of the operation LeftRotate in [4]. Although a "real" implementation will be even more complex, readers might have troubles to understand the effects of this piece of pseudocode.

In contrast, the PROGRES specification of RotateRR, shown in figure 3 above[2], consists of a *production* which is very similar to the graphical explanation of LeftRotate given in [4]. It clearly states that RotateRR reorganizes a subtree with root x such that its red right child y becomes the root of the reorganized subtree. The production's left-hand side (above the separator "::=") defines the graph pattern which has to be transformed. It matches a subgraph of a given host graph, which consists of the given node b = n (the bold node 6 of the input host graph in figure 3), its parent y (node 4), and the parent x of y (node 2). Both the nodes y and b have to be red right children of their parents. Furthermore, the node y may or may not have a black left child a. In our case, the dashed node a matches the node 3 of the input graph, but the production would also be applicable to a red/black tree without the node 3.

2) The figure contains three screen dumps of PROGRES environment tools; cf. section .



```
test FindInconsistencies ( out S: RBNODE [1:n] ) =

            not balanced or not correctColoured

                         'X  : RBNODE

   return S := 'X;
end;
```

Fig. 4: Consistency checking test operation.

The production's right-hand side (below "::=") defines the subgraph which replaces the selected match of its left-hand side. It consists of four nodes with inscriptions of the form v' = 'v. This means that the production preserves all matched nodes including their old intrinsic attribute values and all adjacent edges, which are not explicitly mentioned in the production's left-hand side (as for instance the black left edge from 2 to 1). In the general case, a production deletes nodes and edges, which are part of its left-hand side but not of its right-hand side, and it creates nodes and edges, which are part of its right-hand side but not of its left-hand side. In our case, a number of edges are deleted and created such that x (node 2) becomes the red left child of y (node 4), b (node 6) becomes the red right child of y, and a (node 3) becomes the black, right child of x. Furthermore, the line starting with "embedding" redirects any incoming black left or right edge from the old root node x to the new root node y, i.e. it maintains the embedding of the transformed subtree in its context (cf. output graph in fig. 3). Afterwards, the production's return parameter receives its new value. Finally we have to mention that the explicitly defined graph transformation has an invisible side effect. It marks all affected derived attribute and relation instances in the host graph as being invalid (their new values will be computed upon demand).

All remaining tree modifying operations may be defined in a similar way and are part of the paper's appendix. Before turning our interest to the subject of programming with graph rewrite rules (productions), we will have a closer look onto an additional operation, which has been specified for debugging purposes in figure 5. It is a so-called *subgraph test*, which determines the roots of all inconsistent subtrees of a red/black tree and keeps the given host graph unmodified. It returns either a nonempty set of inconsistent subtree roots or fails. An inconsistent subtree root is a node which violates the restriction balanced or the restriction correctColored (cf. fig. 5).

Based on all the above mentioned subgraph searching and rewriting operations, we can now specify the interface operations of red/black trees as more or less complex *graph rewriting programs*. Please note that (almost) all declared tests and productions search for subgraph patterns, which may either exist or not. In general, a given left-hand side may even match more than one subgraph within a host graph. In that case, the production selects and replaces one of these subgraphs nondeterministically. The crucial point is now that "wrong" selections may lead the overall graph transformation process into dead-ends, i.e. they may prevent the successful execution of other productions later on. To escape out of these dead-ends, PROGRES supports *backtracking*, i.e. our execution machinery is able to undo the effects of unrestricted long sequences of (sub-)graph replacements and to restart the application of a certain production with another match of its left-hand side. Backtracking may even be necessary in our running example (although all tests and productions are deterministic), since the transaction Insert of figure 5 contains a code fragment of the form

choose p1 & p2 else p3 end .

Its execution proceeds as follows:

- Apply production p1 to the given input graph. If p1 fails, then exit from the first branch of the choose-statement and enter its second branch.
- After a successful application of p1 try to execute p2. If p2 succeeds, then return the new host graph as the result of the graph rewriting program, otherwise undo the effects of p1 and enter the choose-statement's else-branch.
- If and only if the execution of p1 & p2 fails, then try to execute p3. The result of p3 (including the case of failure) is now the result of the whole choose-statement.

Beside the above explained concatenation & of subprograms and the choose-statement, a loop- and an or-statement have been used to implement operation Insert. The loop-statement executes its body as often as possible. In our case, it descends from the tree's root to one of its leafs and pushes thereby black edges down the tree as far as possible (production Split). The loop terminates as soon as reorganizations of the tree are no longer possible and the current node n

```
transaction Insert ( Val: integer ) =
   use n: RBNODE do
        GetRoot ( out n )
      & loop
            choose
                 Split ( n, out n )
               & Rotate ( n, out n )
            else
               n := n.down ( Val )
            end
         end
      & (    InsertRight ( n, Val, out n )
          or InsertLeft ( n, Val, out n ) )
      & Rotate ( n, out n )
   end
end;

path down ( Val: integer ) : RBNODE [0:1] -> RBNODE [0:1] =
   [ valid (Val < self.Value) ? =left=>
   | valid (Val > self.Value) ? =right=> ]
end;
```

Fig. 5: Specification of the operation Insert (node in red/black tree).

(1) has a key attribute value equal to Val, or
(2) has a key attribute value less than Val, but not a right child, or
(3) has a key attribute value greater than Val, but not a left child.

These cases are recognized within the path declaration down of figure 5. Its application to the node referenced via variable n fails in all the above mentioned cases, and it returns its left or right child otherwise. The left chiild is selected if and only if the condition of the first branch of "[... | ...]" is true (between "[" and "?"), the right child if and only if the condition of the second branch evaluates to true (between "|" and "?").

Afterwards,execution proceeds with InsertLeft or InsertRight. The application of InsertRight to a given host graph succeeds in case (2) above, and the application of InsertLeft in case (3) above. The usage of an or-statement instead of a choose-statement leaves the decision to the PROGRES runtime system, which production to try first. In case (1), both productions fail, the execution of the whole transaction has to be aborted, and all already issued graph transformations have to be undone. By the way: Graph rewriting programs are termed "transactions", since they are *atomic* (they succeed or fail as a whole) and maintain their host graphs always in a *consistent* state with respect to their graph schemas. The usually mentioned additional properties of transactions, *isolation* and *durability*, are irrelevant in the case of PROGRES, since parallel programming is not supported, and backtracking may cancel the effects of already successfully completed (committed) transactions (for further details about graph rewriting transactions see [29]).

The appendix of the paper contains all remaining declarations of productions and transactions (like InsertRight/Left or Rotate), which were necessary to implement the operation Insert. Compared with an equivalent piece of C or Modula-2 code, these declarations are without any doubts more succinct and more comprehensible. Furthermore, the higher level of "programming" and a very rigid set of type checking rules[3] reduce the probability of errors considerably. There are for instance type checking rules which guarantee for the production RotateRR of figure 3 that

- compatible declarations of all used identifiers are existent,
- sources and target of all edges in its left- and right-hand side belong to correct classes,
- bl and br edges in the embedding part may have x and y as sources,
- the out parameter nn receives a uniquely defined value, and
- assigning the value of n to nn is not in conflict with their type definitions.

Remains the question, whether such a PROGRES specification is just a readable documentation of an efficiently working (manual) implementation or whether it may be used (at least) as a *prototype of the implementation* itself. We will discuss this topic together with the presentation of the PROGRES programming environment within the next section.

3) The language report [25] defines about 300 type checking rules as predicate logic formulas.

3. The PROGRES Environment

Equipped with some basic knowledge about the language PROGRES, the reader probably can imagine the difficulties we had to face during the development of its integrated programming environment. Nevertheless, a 400.000 lines of code large prototype is now available for Sun workstations[4]. It consists of the following *integrated set of tools*:

- a mixed textual/graphical syntax-directed editor together with an incrementally working table-driven pretty-printer, called unparser, and a layout editor,
- an integrated (micro-)emacs like text editor together with an incrementally working LALR-parser,
- an incrementally working type-checker which detects all inconsistencies with respect to the PROGRES language's static semantics and explains highlighted errors,
- an import/export interface to plain text editors and text-processing systems (currently, for text processing system FrameMaker from Frame Technology Corporation only),
- a browsing tool for finding declarations or applied occurrences of given identifiers, highlighting yet incomplete specification parts or unused declarations, and for displaying inferred type information,
- a browser which is an instantiation of the generic graph browser EDGE [23] from the university of Karlsruhe and displays graph schemes in an ER-like fashion,
- an integrated interpreter which translates PROGRES specifications incrementally into intermediate code and forwards this code to an abstract graph rewriting machinery,
- a graph-browser for monitoring host graphs during an interpreter session,
- two compiler backends which translate PROGRES intermediate code into plain Modula-2 and C code, and finally
- tools for version management and three-way merging of different versions of one specification document (for further details see [31, 32]).

The most important basic component of the PROGRES environment is the database system GRAS. It is a *nonstandard database management system* with a graph-oriented data model and a multiple client/server architecture [18]. It supports efficient manipulation of persistent graph structures, incremental evaluation of derived graph properties, nested transactions, undo&redo of arbitrarily long sequences of already committed transactions, recovery from system crashes, and has various options of how to control access of multiple clients to their data structures. In this way, persistency of tool activities including undo&redo and recovery comes for free. And integration of tools is facilitated by storing all data within a GRAS database as a set of related graphs. A single process version of GRAS is currently available as free software (including all sources etc.)[5]. The new multiple client/server version of GRAS, already used within the current PROGRES prototype, will be released soon.

The PROGRES *editor* and *analyzer* are those tools of our environment that assist its users when creating and modifying (hopefully) correct specifications. Both tools are tightly coupled and they are even integrated with the PROGRES interpreter. In this way, the tedious edit/compile/link/debug cycle is avoided and the environment's user is allowed to switch between editing, analyzing, and debugging activities back and forth. The PROGRES editor itself is not a monolithic tool but consists of a number of integrated subtools. These subtools support syntax-directed editing as well as text-oriented editing of specifications, pretty-printing (unparsing), manual rearrangement of text and graphic elements, and finally browsing and searching activities. Figure 6 reveals that PROGRES tools store specifications in the form of two closely related graphs. The so-called *logical graph* contains a specification's abstract syntax tree and all inferred type checking results. The accompanying *representation graph* captures all concrete syntax information, including the chosen layout of (nested) text and graphic fragments. An incrementally working unparser propagates updates of the logical graph into its representation graph. Its counterpart is an incrementally working parser, which propagates changes in the reverse direction. Currently, "free" editing of graphical specification fragments is only supported via a generated textual "ersatz" representation.

The user of the PROGRES environment has two alternatives how to animate a given specification. The first one is based on *direct interpretation*. It is mainly used for debugging purposes, when intertwining of editing, analyzing, and execution activities is advantageous. The second one is to *compile* a specification into

4) Via anonymous ftp from ftp.informatik.rwth-aachen.de in directory pub/unix/PROGRES.

5) Via anonymous ftp from ftp.informatik.rwth-aachen.de in directory pub/unix/GRAS.

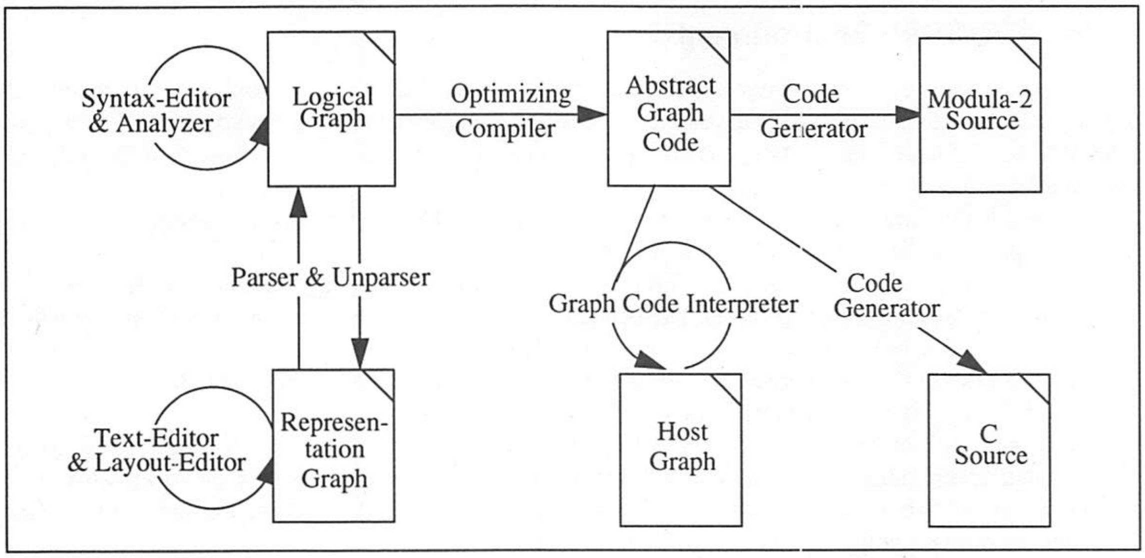


Fig. 6: Data structures and transformation processes in PROGRES.

equivalent Modula-2 or C code and to use the generated source code together with the graph DBMS GRAS as (a rapid prototype of) an abstract data type implementation in a larger program. This program may contain additional components for the realization of a user interface, interprocess communication etc. In both cases, the execution process is divided into two phases:

(1) The initialization phase creates an empty host graph in GRAS, processes all graph scheme declarations and translates them into an internal format with a static dependency graph as its main component. GRAS needs this dependency graph to maintain all derived data (on demand) in a consistent state.

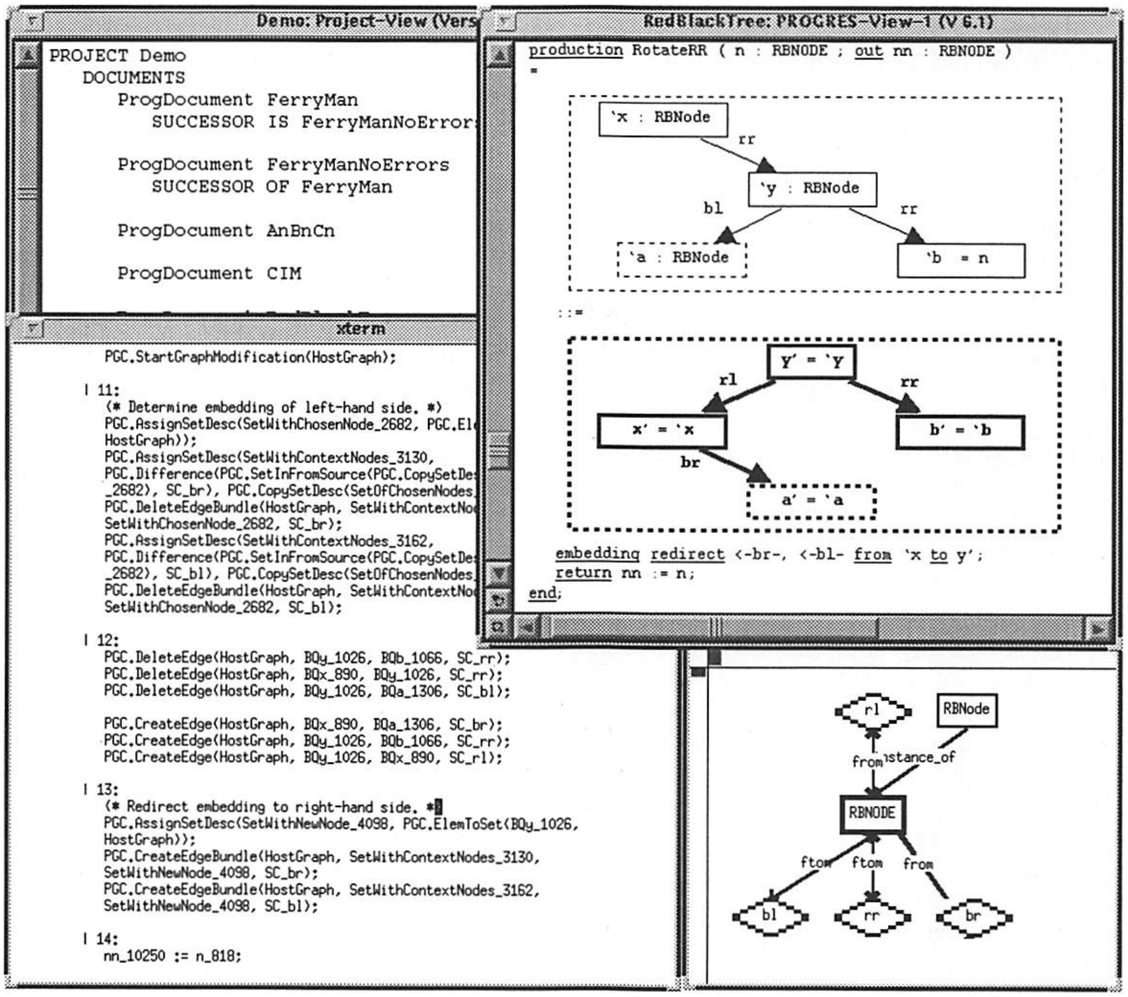
(2) Afterwards, arbitrary productions or transactions may be executed step by step, and they may even be modified and recompiled while the execution process is running.

The diagram in figure 6 gives an overview of the main components and data structures which are involved in the above mentioned activities. It reveals that the editor and analyzer play the role of a conventional compiler's front-end and provide all information about a specification's underlying abstract syntax tree and its static semantics. An *incrementally working compiler* takes this information as input and translates executable increments (on demand) into intermediate code for an abstract graph rewriting machine. The compiler is the most important component of the whole execution machinery and determines its efficiency to a great extent. Especially in the case of productions, where we have to choose between a large number of different intermediate code sequences, which perform the required subgraph matching but differ with respect to runtime efficiency considerably[6] (for further details cf. [34]).

The produced abstract graph code may be executed directly or serves as input for two compiler backends, which produce equivalent Modula-2 or C code. The underlying *abstract graph rewriting machine* combines the functionality of a conventional stack machine à la P-code [24] with backtracking capabilities à la Warren Abstract Machine [30]. Furthermore, its main component offers facilities for manipulating persistent graph structures stored in the database system GRAS.

The translation of abstract graph code into *readable Modula-2 or C source code* is rather straightforward with the exception of backtracking, which requires reversing a program's control flow, restoring old variable values, and undoing graph modifications. Using undo&redo services of GRAS, the main problem is to reverse a conventional program's flow of control without having access to the internal details of its compiler and runtime system. A description of the problem's solution is beyond the scope of this paper but may be found in [33]. It follows the lines of a proposal presented in [20] how to embed backtracking constructs into conventional C programs.

6) Varying from O(n) to $O(n*m^n)$ where n is the number of nodes in a production's left-hand side and m is the number of nodes in a given host graph.



```
PROJECT Demo
   DOCUMENTS
      ProgDocument FerryMan
         SUCCESSOR IS FerryManNoError
      ProgDocument FerryManNoErrors
         SUCCESSOR OF FerryMan
      ProgDocument AnBnCn
      ProgDocument CIM
```

```
production RotateRR ( n : RBNODE ; out nn : RBNODE )
=
  embedding redirect <-br-, <-bl- from 'x to y';
  return nn := n;
end;
```

```
   PGC.StartGraphModification(HostGraph);

 | 11:
   (* Determine embedding of left-hand side. *)
   PGC.AssignSetDesc(SetWithChosenNode_2682, PGC.El
   HostGraph));
   PGC.AssignSetDesc(SetWithContextNodes_3130,
   PGC.Difference(PGC.SetInFromSource(PGC.CopySetDe
   _2682), SC_br), PGC.CopySetDesc(SetOfChosenNodes
   PGC.DeleteEdgeBundle(HostGraph, SetWithContextNo
   SetWithChosenNode_2682, SC_br);
   PGC.AssignSetDesc(SetWithContextNodes_3162,
   PGC.Difference(PGC.SetInFromSource(PGC.CopySetDe
   _2682), SC_bl), PGC.CopySetDesc(SetOfChosenNodes
   PGC.DeleteEdgeBundle(HostGraph, SetWithContextNo
   SetWithChosenNode_2682, SC_bl);

 | 12:
   PGC.DeleteEdge(HostGraph, BQy_1026, BQb_1066, SC_rr);
   PGC.DeleteEdge(HostGraph, BQx_890, BQy_1026, SC_rr);
   PGC.DeleteEdge(HostGraph, BQy_1026, BQa_1306, SC_bl);

   PGC.CreateEdge(HostGraph, BQx_890, BQa_1306, SC_br);
   PGC.CreateEdge(HostGraph, BQy_1026, BQb_1066, SC_rr);
   PGC.CreateEdge(HostGraph, BQy_1026, BQx_890, SC_rl);

 | 13:
   (* Redirect embedding to right-hand side. *
   PGC.AssignSetDesc(SetWithNewNode_4098, PGC.ElemToSet(BQy_1026,
   HostGraph));
   PGC.CreateEdgeBundle(HostGraph, SetWithContextNodes_3130,
   SetWithNewNode_4098, SC_br);
   PGC.CreateEdgeBundle(HostGraph, SetWithContextNodes_3162,
   SetWithNewNode_4098, SC_bl);

 | 14:
   nn_10250 := n_818;
```

Fig. 7: Graph schema, generated Modula-2 code for RotateRR (cutout), etc.

Figure 3, another screen dump of the PROGRES environment, shows a cutout of the generated Modula-2 code for the production RotateRR. It is that fraction of code which modifies the host graph, after a match of the production's left-hand side has been found. It is divided into a number of reexecutable blocks of code (by means of a surrounding case statement), which is a prerequisite for backtracking into already executed Modula-2 code[7].

Compiled PROGRES specifications are about a factor of 10 faster than interpreted abstract graph code. The reasons for such an unusually low interpretation overhead are twofold:

- Parsing, analyzing, and intermediate code generation is performed once and for all, and interpretation is done on the level of optimized abstract graph code only.
- The main speedup is gained when processing conventional language elements like function calls, expression evaluation etc. In all other cases, the overhead for decoding abstract graph code is negligible compared with the complexity of the evoked GRAS database system operations.

The second point mentioned above is the main reason that even compiled PROGRES code is very slow compared with a conventional imperative programming language implementation. Currently, we are able to execute about ten productions per second. Therefore, we are planning to complement the full-fledged database system GRAS with a "light-weight" main memory implementation of graphs. This implementation might be useful, when persistence of data, recovery, backtracking, and automatic maintenance of derived data is of no importance. This will improve the efficiency of compiled (mini-)PROGRES code by orders of magnitude.

7) The remaining space is used to display the graphical presentation of our graph schema and a list of specifications with their predecessor/successor version relationships.

Finally, we have to announce that work is underway to "enhance" generated C code with generated tk/tcl *user interface definitions*. Thus, a future version of the PROGRES environment might be used to implement (rapid prototypes of) graph manipulating applications including an easily extendible and refinable user interface.

4. Related Work

When comparing the functionality of programming languages and their environments, it is often difficult to distinguish between properties of a language and properties of its accompanying tools. Therefore, we will not try to discuss related work about very high level languages and their environments separate from each other, and we will even focus our main interest onto those approaches where both a *language and an environment* are existent. Another problem for a comparison of PROGRES with other approaches stems from the fact that PROGRES is a multi-paradigm language which shares at least some properties with a large fraction of all existing programming languages. In order to be able to keep the size of this section reasonable, we will concentrate our interest on *visual programming or specification languages*, which are either rule-oriented or offer at least pattern matching constructs, and which are built upon a more or less graph-like data model. Therefore, all so-called "graph rewriting" systems are omitted which are more or less just term rewriting systems with sharing of subterms (as for instance [13, 17]).

Using these criteria, a surveyable number of visual languages remains for inspection. Compared with PROGRES their deficiencies come from the following sources: in many cases these languages focus on *manipulation of data structures only* and data definition sublanguages are not provided [3, 15, 19]. And even systems like VAMPIRE with its class hierarchies and icon rewriting rules [21] and PAGG with its node type definitions and graph rewriting rules [11] come without a rigid type concept and *without any type checking rules and tools*. Therefore, all these systems postpone recognition of programming errors to runtime and suffer from the same disadvantages as any typeless or weakly typed programming language.

With respect to its graph schema definition capabilities, PROGRES is more similar to so-called visual database query or programming languages [1, 2, 5, 7]. But these languages have less expressive pattern matching and replacing constructs (no object set patterns etc.), and some of them are even not computational complete (no recursion etc.). Furthermore, neither the above mentioned languages nor, to the best of our knowledge, any other diagrammatic rule-oriented language offers *nondeterministically* working control structures together with the ability to *backtrack* out of dead-ends of locally failing rewriting processes.

5. Summary

This paper contains a brief presentation of the *very high level language* PROGRES which provides its users with diagrammatic as well as textual constructs for the definition of graph schemas, derived graph properties, graph rewrite rules, and complex graph transformation processes. The accompanying PROGRES environment offers means to create, analyze, and interpret specifications of graph-like data types. It contains even two (prototypes of) compilers which translate a given PROGRES specification into equivalent Modula-2 or C code. Being an integrated set of tools with support for intertwining these activities, PROGRES combines the flexibility of interpreted languages with the safeness of compiled (statically typed) languages and their environments.

The current PROGRES implementation is built on top of our own free software database system GRAS and the user interface toolkit Interviews from Stanford University. It has a size of about 400,000 lines of code and is (unfortunately) partially written in C, Modula-2, and C^{++}. A considerable amount of code is even not hand-written but generated by means of a meta-programming environment (which is the result of a related software engineering environment project IPSEN; cf. [8, 22]). Therefore, generating, compiling, and linking a new PROGRES release is a kind of research topic in its own respect. As a consequence, the public release of the PROGRES environment consists of binary code (for Sun workstations) only, and its (meta-)sources are not delivered up to now. Both the PROGRES environment and its database system GRAS are available via anonymous ftp from ftp.informatik.rwth-aachen.de. The author of this paper may be contacted for more detailed information about future releases etc.

References

[1] Andries M., Engels G.: *Syntax and Semantics of Hybrid Database Languages*, in: [10]

[2] Angelaccio M., Catarci T., Santucci G.: *QBD*: A Graphical Query Language with Recursion*, in: IEEE Transactions on Software Engineering, vol. 16, no. 10, Los Alamitos: IEEE Computer Society Press (1990), 1150-1163

[3] Arefi F., Hughs Ch.E., Workman D.A.: *Automatically Generating Visual Syntax-Directed Editors*, in: CACM, vol. 33, no. 3, New York: acm Press (1990), 349-360

[4] Cormen Th.H., Leiserson Ch.E., Rivest R.L.: *Introduction to Algorithms*, Cambridge: MIT Press (1990)

[5] Czejdo B., Elmasri R., Rusinkiewicz M., Embley D.W.: *A Graphical Data Manipulation Language for an Extended Entity-Relationship Model*, in: IEEE Computer, vol. 23, no. 3, Los Alamitos: IEEE Computer Society Press (1990), 26-37

[6] Consens M., Mendelzon A.: *Hy+: A Hygraph-based Query and Visualization System*, in: Buneman P., Jajodia S. (eds.): *Proc. 1993 ACM SIGMOD Conf. on Management of Data*, SIGMOD RECORD, vol. 22, no. 2, New York: acm Press (1993), 511-516

[7] Catarci T., Santucci G.: *Query By Diagram: A Graphic Query System*, in: *Entity-Relationship Approach*, Amsterdam: Elsevier Science Publ. B.V. (1989), 291-308

[8] Engels G., Lewerentz C., Nagl M., Schäfer W., Schürr A.: *Building Integrated Software Development Environments Part I: Tool Specification*, in: acm Transactions on Software Engineering and Methodology, vol. 1, no. 2, New York: acm Press (1992), 135-167

[9] Ehrig H., Kreowski H.-J. (eds.): *Proc. 4th Int. Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 532, Berlin: Springer Verlag (1990),

[10] Ehrig H., Schneider H.-J.: *Proc. Dagstuhl-Seminar 9301 on Graph Transformations in Computer Science*, LNCS 776, Berlin: Springer Verlag (1994)

[11] Göttler H., Günther J., Nieskens G.: *Use Graph Grammars to Design CAD-Systems*, in [9], 396-410

[12] Glinert E.P.: *Visual Programming Environments: Paradigms and Systems*, Los Alamitos: IEEE Computer Society Press (1990)

[13] Glauert J.R., Kennaway J.R., Sleep M.R.: *Dactl: An Experimental Graph Rewriting Language*, in: [9], 378-395

[14] Gemis M., Paredaens J., Thyssens I., Van den Bussche J.: *GOOD: A Graph-Oriented Object Database System*, in: Buneman P., Jajodia S. (eds.): *Proc. 1993 ACM SIGMOD Conf. on Management of Data*, SIGMOD RECORD, vol. 22, no. 2, New York: acm Press (1993), 505-510

[15] Himsolt M.: *GraphED: An Interactive Graph Editor*, in: Proc. STACS 89, LNCS 349, Berlin: Springer Verlag (1988), 532-533

[16] Hudson S.E., King R.: *The Cactis Project: Database Support for Software Environments*, IEEE Transactions on Software Engineering, Vol. 14, No. 6, Los Alamitos: IEEE Computer Society Press (1988)

[17] Kuchen H., Loogen R., Moreno-Navarro J.J., Rodriguez-Artalejo M.: *Graph-based Implementation of a Functional Logic Language*, in: Proc. ESOP '90, LNCS 432, Berlin: Springer Verlag (1990), 271-290

[18] Kiesel N., Schürr A., Westfechtel B.: *Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications*, in: CASE '93 Proc. 6th Int Workshop on Computer-Aided Software Engineering, Los Alamitos: IEEE Computer Society Press (1993), 272-286

[19] Löwe M., Beyer M.: *AGG - An Implementation of Algebraic Graph Rewriting*, in: Proc. 5th Int. Conf. on Rewriting Techniques and Applications, LNCS 690, Berlin: Springer Verlag (1993), 451-456

[20] Liu Y.: *btc: A Backtracking Procedural Language*, Technical Report 203, University of Queensland, Key Center for Software Technology, Australia (1991)

[21] McIntyre D.W., Glinert E.P.: *Visual Tools for Generating Iconic Programming Environments*, in: [35], 162-168

[22] Nagl M.: *A Characterization of the IPSEN-Project*, in: Proc. Int. Conf. on System Development Environments & Factories, Pittman (1990) 141-150

[23] Newbery Paulisch F.: *The Design of an Extendible Graph Editor*, LNCS 704, Berlin: Springer Verlag

[24] Pemberton St., Daniels M.: *Pascal Implementation: The P4 Compiler*, Ellis Horwood Ltd. (1982)

[25] Schürr A.: *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*, Ph.D. Thesis, RWTH Aachen, Wiesbaden: Deutscher Universitätsverlag (1991)

[26] Schürr A.: *Logic Based Structure Rewriting Systems*, in: [10], 341-357

[27] Schürr A.: *Programmed Logic Based Structure Rewriting System*, TR 94-12, Aachener Informatik-Berichte, RWTH Aachen, Germany

[28] Schwartz J., Westfechtel B.: *Integrated Data Managment in a Heterogeneous CIM Environment*, in: Proc. CompEuro '93, Los Alamitos: IEEE Computer Society Press, 272-286

[29] Schürr A., Zündorf A.: *Nondeterministic Control Structures for Graph Rewriting Systems*, in: Proc. WG'91 Workshop in Graphtheoretic Concepts in Computer Science, LNCS 570, Berlin: Springer Verlag (1992), 48-62

[30] Warren D.H.D.: *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, Menlo Park

[31] Westfechtel B.: *Revision Control in an Integrated Sofware Development Environment*, in: Proc. 2nd Int. Workshop on Software Configuration Management, ACM Software Engineering Notes, vol. 17, no. 7, New York: acm Press (1989), 96-105

[32] Westfechtel B.: *Structure-Oriented Merging of Revisions of Software Documents*, in: Proc. 3rd Int. Workshop on Software Configuration Management, New York: acm Press (1991), 68-80

[33] Zündorf A.: *Implementation of the Imperative/Rule Based Language PROGRES*, Technical Report AIB 92-38, RWTH Aachen, Germany (1992)

[34] Zündorf A.: *A Heuristic Solution for the (Sub-)Graph Isomorphism Problem in Executing PROGRES*, Technical Report AIB 93-5, RWTH Aachen

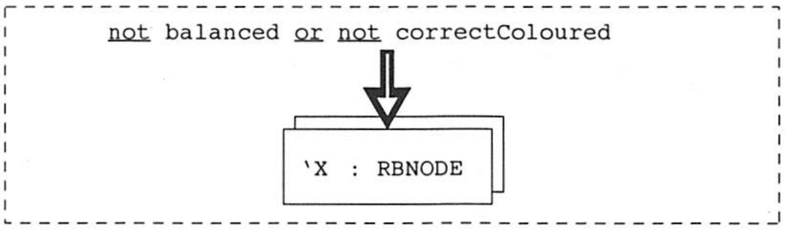
[35] Proc. of the 1992 IEEE Workshop on Visual Languages, Los Alamitos: IEEE Computer Society Press (1992)

Author Biography

Andy Schürr is a Bavarian inhabitant of Germany. He got his master degree in computer science (with mathematics as subsidiary subject) from the Technical University of Munich (Germany) in 1986. He finished his Ph.D. thesis at the RWTH Aachen (Germany) in 1992 and is now a "wissenschaftlicher Assistent" at the Lehrstuhl für Informatik III, RWTH Aachen (more or less a kind of assistent professor). He is a member of the ACM and his main interests include software engineering environments, nonstandard database systems, very high level visual programming languages, and especially graph rewriting systems.

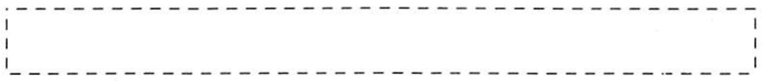
Appendix

```
spec RedBlackTree;

   node class RBNODE
      intrinsic
         key Value : integer;
      derived
         Height : integer = max ( self.lHeight, self.rHeight );
         lHeight = [ self.-bl->.Height + 1 | self.-rl->.Height | 0 ];
         rHeight = [ self.-br->.Height + 1 | self.-rr->.Height | 0 ];
   end;

   edge type bl : RBNODE [0:1] -> RBNODE [0:1];

   edge type br : RBNODE [0:1] -> RBNODE [0:1];

   edge type rl : RBNODE [0:1] -> RBNODE [0:1];

   edge type rr : RBNODE [0:1] -> RBNODE [0:1];

   static path left : RBNODE [0:1] -> RBNODE [0:1] =
      [ -bl-> | -rl-> ]
   end;

   path right : RBNODE [0:1] -> RBNODE [0:1] =
      [ -br-> | -rr-> ]
   end;

   path parent : RBNODE [0:1] -> RBNODE [0:1] =
      [ <=left= | <=right= ]
   end;

   restriction balanced : RBNODE =
      valid (self.lHeight = self.rHeight)
   end;

   restriction correctColoured : RBNODE =
      not with ((-rl-> or -rr->) & (-rl-> or -rr->))
   end;

   test FindInconsistencies ( out S: RBNODE [1:n] ) =
```

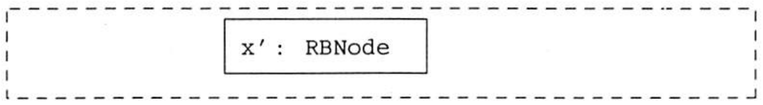


```
      return S := 'X;
   end;

   path down ( Val: integer ) : RBNODE -> RBNODE [0:1] =
      [ valid (Val < self.Value) ? =left=>
      | valid (Val > self.Value) ? =right=> ]
   end;

   node type RBNode : RBNODE end;

   function max : ( x: integer ; y: integer ) -> integer =
      [ x <= y ? y | x ]
   end;

   transaction MAIN =
      use i := 1
      do
           InsertFirst ( 4 )
         & loop
                Insert ( i )
              & i := (i + 1)
           end
      end
   end;
```

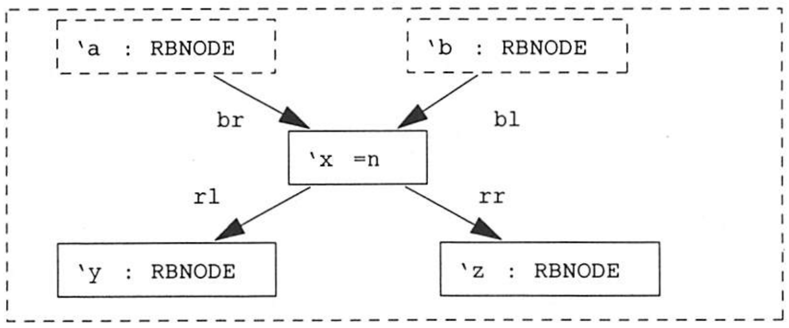
```
production InsertFirst ( Val: integer ) =
```



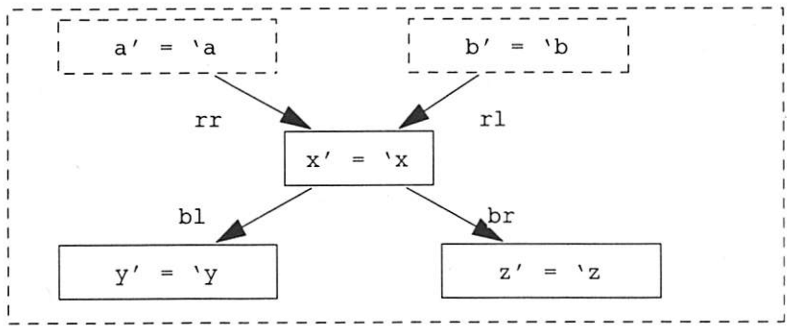
```
   ::=
```



```
   transfer x'.Value := Val;
end;

transaction Insert ( Val: integer ) =
   use n: RBNODE do
        GetRoot ( out n )
      & loop
           choose
                Split ( n, out n )
              & Rotate ( n, out n )
           else
              n := n.down ( Val )
           end
        end
      & (   InsertRight ( n, Val, out n )
         or InsertLeft ( n, Val, out n ) )
      & Rotate ( n, out n )
   end
end;

production Split ( n: RBNODE ; out nn: RBNODE ) =
```

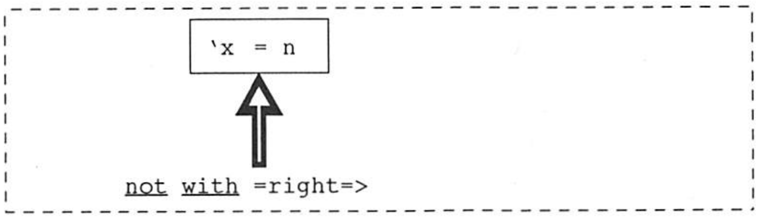


```
   ::=
```

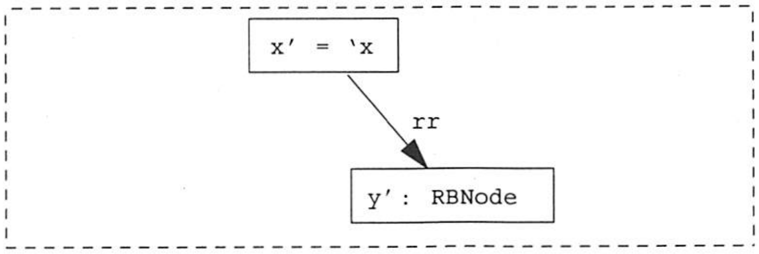


```
   return nn := n;
end;
```

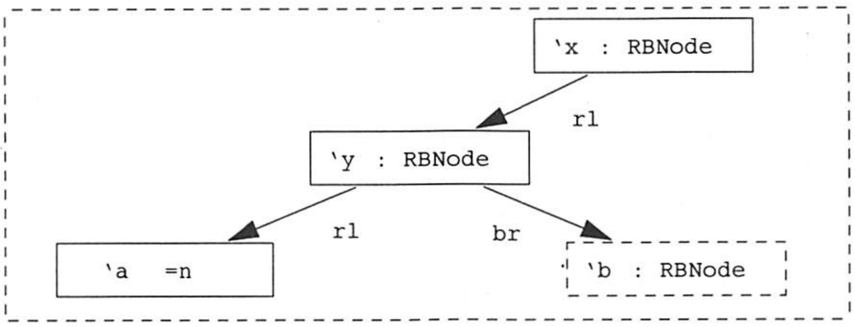
```
production InsertRight ( n: RBNODE ; Val: integer ; out nn: RBNODE ) =
```



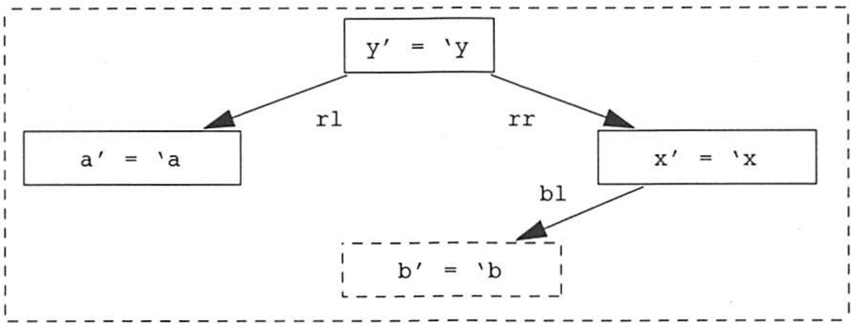
```
   ::=
```



```
   condition Val > 'x.Value;
   transfer y'.Value := Val;
   return nn := y';
end;
production RotateLL ( n : RBNODE ; out nn : RBNODE ) =
```

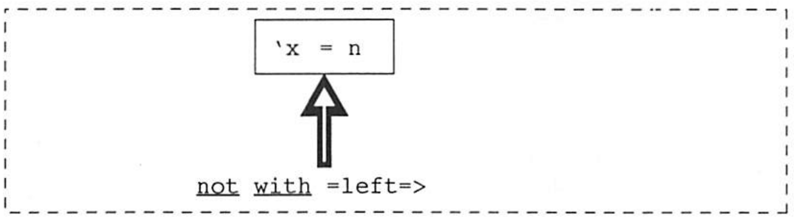


```
   ::=
```

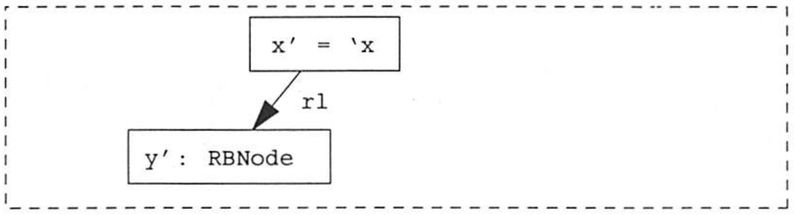


```
   embedding redirect <-br-, <-bl- from 'x to y';
   return nn := n;
end;
```

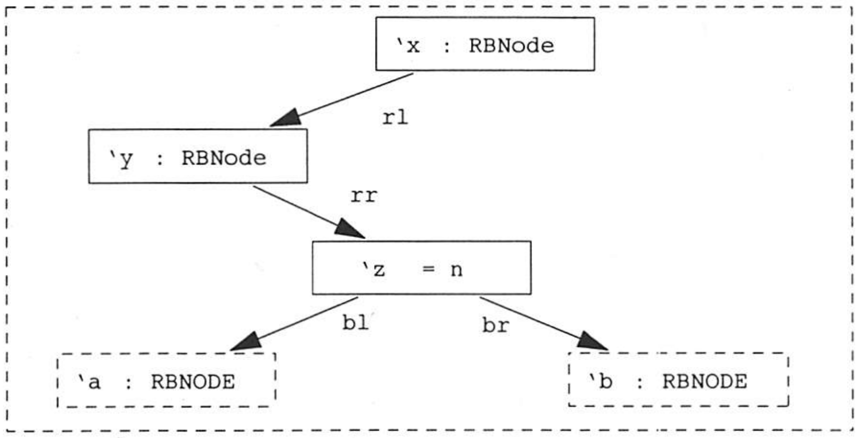
```
production InsertLeft ( n : RBNODE ; Val : integer ; out nn : RBNODE ) =
```



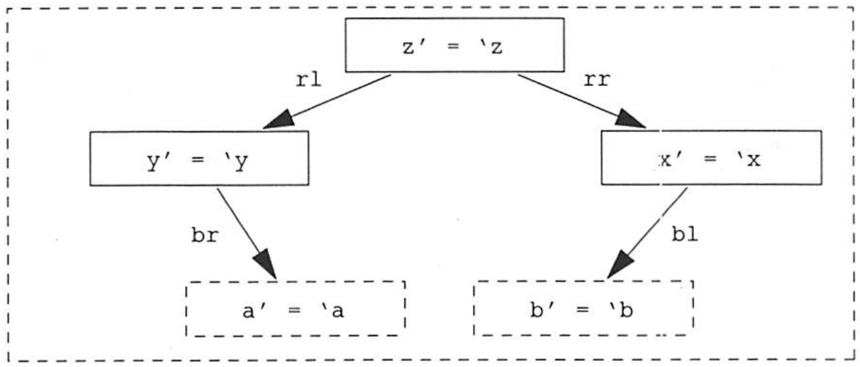
```
    ::=
```



```
    condition Val < 'x.Value;
    transfer y'.Value := Val;
    return nn := y';
  end;
  production RotateLR ( n : RBNODE ; out nn : RBNODE ) =
```

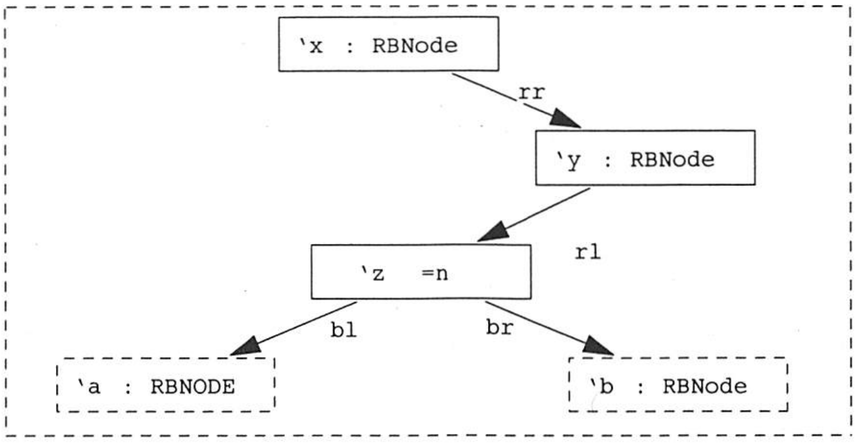


```
    ::=
```

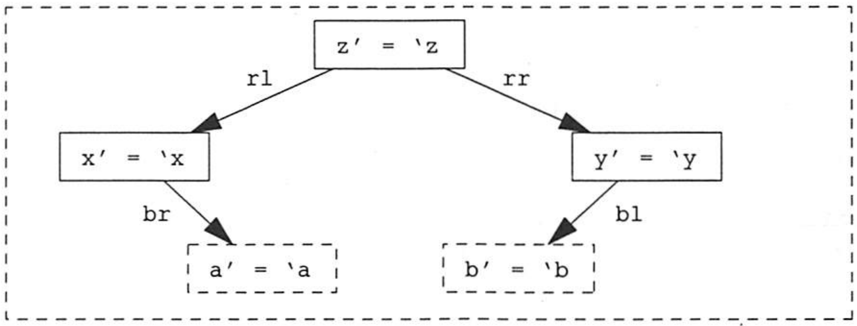


```
    embedding redirect <-br-, <-bl- from 'x to z';
    return nn := y';
  end;
```

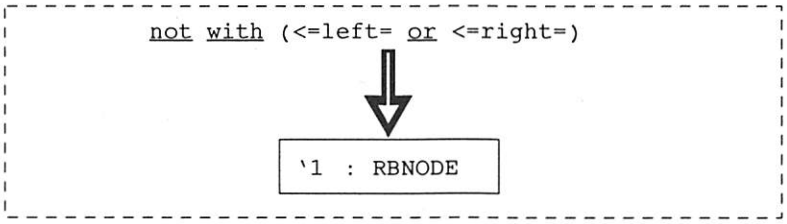
```
production RotateRL ( n : RBNODE ; out nn : RBNODE )
=
```



```
    ::=
```

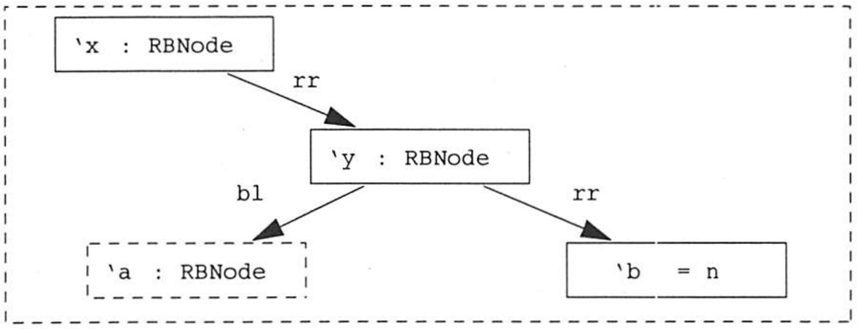


```
    embedding redirect <-br-, <-bl- from 'x to z';
    return nn := y';
end;
test GetRoot ( out nn : RBNODE ) =
```

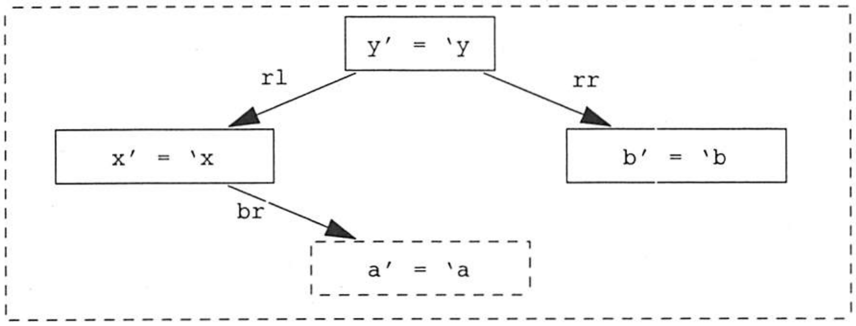


```
end;
transaction Rotate ( n: RBNODE ; out nn: RBNODE ) =
    loop
           RotateLR ( n, out nn )
        or RotateRL ( n, out nn )
        or RotateLL ( n, out nn )
        or RotateRR ( n, out nn )
    end
end;
```

```
    production RotateRR ( n : RBNODE ; out nn : RBNODE ) =
```



```
        ::=
```



```
        embedding redirect <-br-, <-bl- from 'x to y';
        return nn := n;
    end;

end.
```

End-User Systems, Reusability, and High-Level Design

Glenn S. Fowler (gsf@research.att.com)
John J. Snyder (jjs@research.att.com)
Kiem-Phong Vo (kpv@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

During the past ten years, the number of computer users has grown by orders of magnitude. This has been brought about by dramatic increase in computing power combined with equally dramatic decrease in hardware costs. Beyond "stand-alone" user applications like word processing and spreadsheets, new classes of business applications arise where competitive advantage is created by empowering "end-users" with instant access to relevant information. In many cases, code already exists to access and process the desired information; the challenge is finding a way to couple such processing capabilities to individual user requests in a timely, specific, and friendly fashion. The keys to such end-user systems lie in high-level design and software reusability. This paper describes a language tool EASEL (End-User Application System Encoding Language) for building end-user systems and experiences in its development and deployment.

1. Introduction

An end-user is a user who needs to perform some computing functions but may not be well-acquainted with the underlying theories and mechanics. In this sense, even an expert computer user can be an end-user in some application domain. An end-user application is a software system designed to be used by end-users. As such, the primary characteristic of an end-user system is an interface that is easy to use, directly addresses users' needs, and hides any complexity in the solution methods. As computing techniques become more diverse and complex while the cost of computing machinery becomes cheaper, the market for end-user applications gets larger. This is particularly true in business computing where combinations of techniques from networking, databases, statistical analyses, equation solving, and graphics are routinely required.

Though computing techniques can be complex, there are a multitude of high quality software tools readily available to solve such problems. From a system construction point of view, it is desirable to take advantage of such tools in building a new application. Coupled with the requirement of a good user interface, the main challenges in building an end-user application are:

- Designing a system architecture that maps closely to users' needs
- Building an interface that reflects that architecture

- Leveraging as much as possible from existing software tools in implementation.

As successful applications often live long and beget others, additional challenges are to ensure that a system is easily evolvable, and to grow the pool of reusable software tools as systems are built. A sign of a successful software development organization is its ability to build families of applications quickly and still provide good maintenance support. This is possible only if:

- The system construction method encourages building reusable tools
- Enough characteristics of an application family can be abstracted into reusable templates
- A means exists to help the coding and maintenance of such templates.

Over the past 10 years, we have been working with a language and system to build end-user applications called EASEL (End-User Application System Encoding Language). EASEL applications use interactive constructs such as windows, forms, menus, and hypertexts to provide easy to use interface and build or reuse computational tools for other application-specific tasks. Within AT&T, EASEL has been used to build hundreds of applications ranging from simple educational systems to sophisticated applications used to plan and manage the telephone network. Some of these applications are used world-wide and support hundreds of international users on a daily basis. Early experience with EASEL was reported in [Vo90]. This paper describes the current state of the EASEL language, how its approach to system construction encourages software reuse, how language macros help build a higher abstraction levels, and relates some experience in building families of end-user applications with EASEL.

2. End-user system architecture

To see how EASEL helps to build an end-user system, we need to understand the components that comprise such a system. The logical structure of an end-user system can be divided into four layers [Vo90] as shown in Figure 1. The top two layers represent *Design Programming*, i.e., programming activities focusing on the user interface and high-level tasks as seen from the user's point of view. The lower two layers represent *Computation Programming*, i.e., activities focusing on how the high level tasks are to be implemented and with what data structures.

The *User Interface* layer allows users to manipulate the systems using well-defined and easy to use steps. A major function of this layer is to hide all differences and idiosyncrasies in the interfaces of underlying computing tools and techniques. An EASEL application builds this layer using menus, forms, and hypertexts to provide an active interface that guides users in taking appropriate computational steps.

The *System Structure* defines a task partition and relationships among tasks. In the EASEL framework, the design process of an end-user system begins with identification of tasks and subtasks as seen from the user's perspective. The high level tasks are mapped to objects known as *frames* which are interconnected in a *frame network*. Each frame defines all user

Design Programming	User Interface	Forms, Menus, Text
	System Structure	Tasks and Subtasks
Computation Programming	Computational Functions	Application Specific Code
	Data Architecture	Application Specific Data Types

Figure 1: Interactive End-User Systems Architecture

dialogs appropriate to the task and prescriptions for computing methods necessary to carry out that task. Dialogs can cause transitions to other frames in the frame network or induce certain low level computations. For example, a simple electronic mail application might consist of the following tasks:

- Getting a list of users,
- Picking addressees from the user list,
- Filling out a mail form, and
- Sending the mail.

Each of these tasks maps to a frame that defines:

- Its own set of relevant parameters (e.g., login ids, mail headers, etc.),
- Mechanisms to obtain the parameters (e.g., from the user, from login database, etc.), and
- Appropriate commands to drive lower-level computational processes (e.g., running the `mail` command).

Thus, frames focus on high level activities required to perform a given task. Actual computations are performed at a lower level by invoking applications code, utilities, and other packages.

The *Computational Functions* layer consists of utilities and application-specific code embodying the computational methods to access, transform, and generate the data necessary to accomplish the end-user's tasks as requested. The EASEL language provides for string manipulation, mathematical operations, file input/output, and event handling. In addition, there are several language mechanisms to execute application specific code that may be in C or some interpretive languages such as the UNIX shell [Bou78, BK89] or AWK [AKW88].

The *Data Architecture* layer defines the types of data to be manipulated along with their storage and access methods. Specific data types depend on the application and the tools used to perform the computational functions. Data types may be tuned to support, facilitate, and optimize execution of algorithms and methods at the computational level. Examples include EASEL variables, C and C++ data types and structures, data for relational and object-oriented databases, as well as data types required by specific packages, such as statistical or queuing packages.

The four-layer architecture points out some useful insights on reusability in some broad categories of tools. Traditional UNIX tools are often designed to do single tasks that, in many cases, embody powerful data structures and algorithms. Since their interfaces are geared toward the specific computing methods being implemented, they are not easily usable by end-users although they are immensely reusable. At the other end of the spectrum, many screen-oriented applications based on screen libraries like *curses* [Arn84, Vo85] or *X* [Nye90] and various spreadsheets or database packages are easy to use, but offer little reuse because the computational methods are too intertwined with the implementation of other parts of the system. EASEL is an attempt at bridging this *reuse gap* between general purpose but hard to use tools and application specific but easy to use software.

EASEL's approach to system construction can be summed up as that of *separating Design Programming from Computational Programming.* The EASEL language focuses on expressing the high level design of a system including its user interface and task partition. Most computational details in the lower two layers are left out at this level but enough of the interface to appropriate software components can be specified. In this way, tool reuse is naturally a component of the EASEL's system construction method. As we shall see later, by bringing in a language tool for program design, EASEL also makes possible the reuse of certain high level tasks in much the same way that software tools traditionally improve computational reuse.

3. The EASEL language

The EASEL language is block structured where the block types map to certain high level activities. Some blocks define user interface components and tasks such as:

- `{frame` – a high level task
- `{context` – grouping of related activities
- `{menu` – selections to be decided by users
- `{question` – a group of questions is a form
- `{write` – display text to screen

Other blocks and statements define computations such as:

- `{action` – runs the enclosed shell script

- `{process` – runs named cooperating UNIX process and sends enclosed text to its standard input
- `~Ccall` – calls the named C function
- `~call ~goto ~overlay ~return` – transition among frames
- `~evglobal ~evlocal` – event handling
- file input and output
- string handling
- mathematical operations and functions

Other language statements provide for managing the scope of variables, manipulating the run-time UNIX process environment and keyboard bindings and macros. Another group of statements is used to modify default display attributes, such as window locations, border styles, colors, etc.

EASEL variables need not be declared and are initialized as empty strings. Most operations including those requiring communicating with external processes or subroutines result in string values. However, in cases where numerical values are required, the mathematical assignment statement := can be used to do such computations.

EASEL applications are constructed in a network of frames. Each frame is defined in a frame block:

```
{frame  FrameID  ~arg1 ~arg2 ...
        ....
}f
```

Each frame has its own name or frame id, may accept arguments, may contain statements and other nested blocks, and may return values to its caller. EASEL variable names (and keywords) begin with the ~ (tilde) character.

Composite blocks provide grouping of statements and nested blocks:

```
{context  (entry):(exit)
        ....
}c
```

Control flow as exemplified above is directed via (*entry*) and (*exit*) conditions. A block or statement is executed only if its (*entry*) condition is true; control loops through the block or statement until its (*exit*) condition is satisfied. An omitted, or null, condition is taken as true.

Rather than examining the EASEL programming language in detail, we will illustrate how to build a system with EASEL using a small example.

3.1. An EASEL email application

An electronic mail application serves as a small but realistic example. The main tasks of the application consist of:

- Getting a list of users,
- Asking the user to select addressees,
- Filling out a mail form, and
- Sending the mail.

The design of the frame network for such an *email* application might then look like Figure 2.

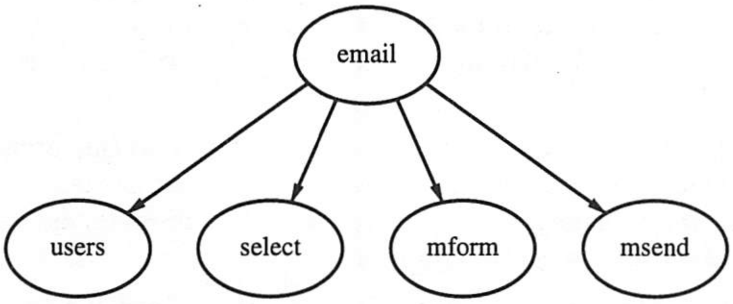


Figure 2: The *email* frame network

Below is the EASEL code for the top level `email` frame. Line 1 begins a frame block for frame `email`. Line 2 calls frame `users` to get a list of users that is assigned to variable `~users`. Line 3 calls frame `select` with argument `~users` to ask user to make selections which will be assigned to variable `~to`. Line 4 calls frame `mform` with argument `~to`. This frame will return three values to be assigned to `~to`, `~subj` and `~mesg`. Line 5 begins with an entry condition that is true only when both `~to` and `~mesg` are not null; if that is the case, then frame `msend` is called to send the message to the selected people. Line 6 denotes the end of the frame block.

```
1: {frame  email
2:     ~call users  > ~users
3:     ~call select ~users  > ~to
4:     ~call mform ~to  > ~to ~subj ~mesg
5:     (~to!=~null && ~mesg!=~null): ~call msend ~to ~subj ~mesg
6: }f
```

The first task or frame called by `email` is `users`. Below is the code for `users`. Line 2 runs `/bin/ksh` as a cooperating process or coprocess. Values returned from the coprocess will be assigned to variable `~logins`. Line 3 specifies the "end-of-response" and "end-of-commands" protocol delimiters to be used between EASEL and the coprocess. When EASEL is done sending data to the coprocess (the text on line 5), it sends the end-of-commands text (`echo ShIsDone\n`). EASEL then starts reading text returned from the coprocess until it sees the end-of-response text (`ShIsDone\n`). Line 4 says that the screen will not be disturbed during this execution so EASEL will not refresh the screen when it gets back control from the coprocess. Line 5 is the complete body of text to be sent to the coprocess; on Suns running NIS (formerly Yellow Pages), this shell script will generate the current list of logins. Line 6 ends the process block. Line 7 causes control to return to the calling frame and returns the value contained in the variable `~logins`.

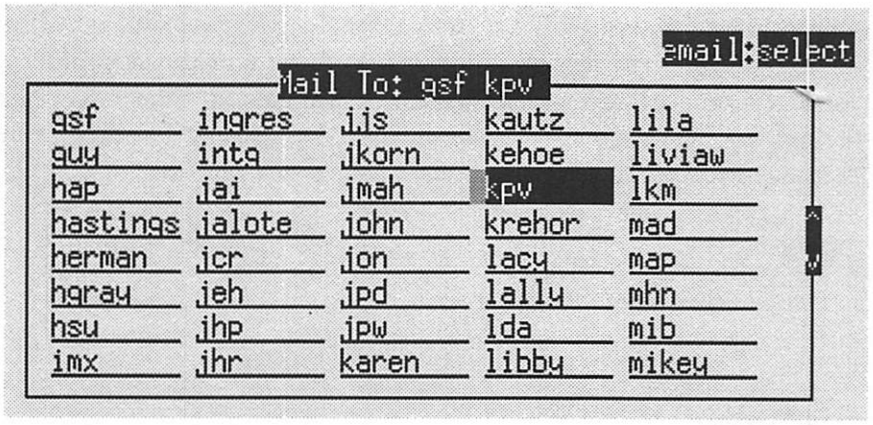
```
1: {frame  users
2:     {process  "/bin/ksh"  > ~logins
3:         ~!"ShIsDone\n","echo ShIsDone\n"
4:         ~$
5:         ypcat passwd | cut -d: -f1 | sort | uniq | xargs
6:     }p
7:     ~return ~logins
8: }f
```

The next task is to ask the user to select the desired addressees from the available list of users. Below is the frame `select` that performs this task. Line 1 begins frame `select`, which has argument `~list`. Lines 2–8 present the `~list` of logins as a menu to the end-user, builds a `~to` list adding each user selection, and loops until a null choice is entered. The `.window` statement on line 3 defines a window for the menu. If this is not defined EASEL will construct some default window whose size and placement are designed to make good use of screen real estate. Line 5 provides the list of options to be shown, namely those stored in variable `~list`. Delimiters for the list are specified as space, tab, or newline characters. Line 6 builds a list of the logins selected by the user and stores the result in the variable `~to`. This variable is used in the menu's title given in line 4 to give immediate feedback to user on the current set of selections. Line 9 returns the value of `~to` to the calling frame.

```
 1: {frame  select ~list
 2:     {menu  :(~pers == ~null)  ~pers
 3:         .window( x=5, xlen=50, y=0, ylen=10 )
 4:         Mail To: ~to
 5:         {option  ~list  " \t\n"
 6:             ~to  = ~to * ~pers * " "
 7:         }o
 8:     }m
 9:     ~return ~to
10: }f
```

A screen snapshot of `select` is shown in Figure 3. Note that the frame stack `email:select` shows the traversal sequence in the frame network to get to the frame `select`. If allowed by system designers, experienced users of an EASEL application can use the commands at the bottom of the screen to directly access any frame in the network.

The third `email` task is to display a mail form and let the user fill it in. Below is the frame `mform`. The context block beginning on line 2 is used to group the enclosed question blocks so they will be displayed together as a unit or form. The `.form` statement on lines 3–8 specifies that answer fields should be shown in color 2 (typically underlining on black and white terminals) with attributes set to printable characters. The `.form` statement also includes a template for the form and indicates which question variables correspond to which answer fields. For example, line 4 indicates that the answer field is the `~To` variable, which, in this case, was passed as an argument to the frame. Thus any menu selections made by

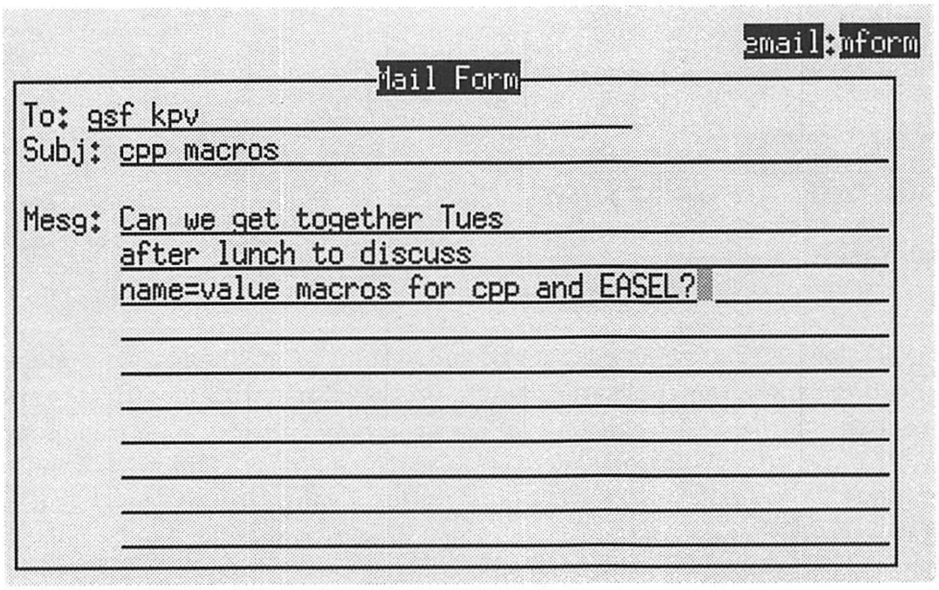


```
                                              email:select
              Mail To: gsf kpv
gsf       ingres   jjs      kautz    lila
guy       intg     jkorn    kehoe    liviaw
hap       jai      jmah     kpv      lkm
hastings  jalote   john     krehor   mad
herman    jcr      jon      lacy     map
hgray     jeh      jpd      lally    mhn
hsu       jhp      jpw      lda      mib
imx       jhr      karen    libby    mikey
```

Figure 3: The `select` screen

the user in the `select` frame will be shown on the form as soon as it is displayed. The user can edit the answer field on the form or move on to the next field. Lines 10–12 provide help text for this form should the user request help. Although not shown here, help text may include EASEL variables as well as hypertext links. Lines 13–22 are the question blocks used to build up the form; each provides a variable to collect the user's response to that question. The `.field` statement in line 20 overrides the suggested template and specifies that the answer `~Mesg` field be `r`epeated 10 times for a ten-line answer. If the user types beyond the 10th line, the answer field will scroll (unless the answer has been restricted to the length specified). Although the syntax for specifying a form is a bit verbose with its `.form` statement and group of question blocks, it saves frame designers from counting row and column coordinates and allows entry conditions on question blocks to tailor field access dynamically. Line 24 returns the values in `~To`, `~Subj`, and `~Mesg` back to the calling frame. Figure 4 shows a screen snapshot of the form.

```
 1: {frame  mform  ~To
 2:     {context
 3:         .form( c=2, a=p,
 4:         To: <____________________ $ ~To
 5:         Subj: <___________________________ $ ~Subj
 6:
 7:         Mesg: <___________________________ $ ~Mesg
 8:         )
 9:         Mail Form
10:         {descript
11:             Some HELP for the form
12:         }d
13:         {question  ~To
14:             To:
15:         }q
16:         {question  ~Subj
17:             Subj:
18:         }q
19:         {question  ~Mesg
20:             .field( r=10 )
21:             Mesg:
22:         }q
23:     }c
24:     ~return ~To ~Subj ~Mesg
25: }f
```


email:mform

Mail Form

To: gsf kpv

Subj: cpp macros

Mesg: Can we get together Tues
after lunch to discuss
name=value macros for cpp and EASEL?

Figure 4: The `mform` screen

The final task, if the user has provided non-null text for the addressees and the message, is to send the user's e-mail, as coded in frame **`msend`** below. Line 2 gets the current value of the environment variable **`$LOGNAME`** and stores it in the EASEL variable **`~LOGNAME`** (for use in the mail message body). Lines 3–14 communicate with the same coprocess **`/bin/ksh`** as was set up in the frame **`users`**. This time, what is being sent to the shell is a multi-line mail command that is parameterized by several EASEL variables. The variable **`~efsdate`** in line 8 is an EASEL read-only variable containing the current date as a string, e.g., **`Sat May 21 01:40:10 EDT 1994`**.

```
 1: {frame  msend  ~To ~Subj ~Mesg
 2:     ~getenv ~LOGNAME
 3:     {process  "/bin/ksh"
 4:         ~!"ShIsDone\n","echo ShIsDone\n"
 5:         ~$
 6:         mail ~To <<!!
 7:         Subj: ~Subj
 8:         Date: ~efsdate
 9:
10:         ~Mesg
11:
12:         Thanks! ~LOGNAME
13:         !!
14:     }p
15: }f
```

The actual text for the mail command, as sent to the shell, is shown below. In prototyping and early testing of a frame, the process block for the mail command in lines 3–14 might be enclosed in a write block (to the screen or to a file) to simply write out the mail command as it would be sent to the coprocess to let the command be checked without sending any unnecessary mail.

```
mail gsf kpv <<!!
Subj: cpp macros
Date: Sat Apr 30 01:40:10 EDT 1994

Can we get together Tues
after lunch to discuss
name=value macros for cpp and Easel?

Thanks! jjs
!!
```

That has been a quick look at EASEL as a language and system to build end-user systems. As the *email* application shows, the application design process starts with identifying tasks from user's perspective. These tasks can be quickly prototyped in the EASEL language without lower level computations. In this way, the tasks can be tested out with users to see if they are suitable. In parallel to or with user testing, computational tasks can be implemented and tested. This cycle continues until the system is complete. Figure 5 shows this system construction method.

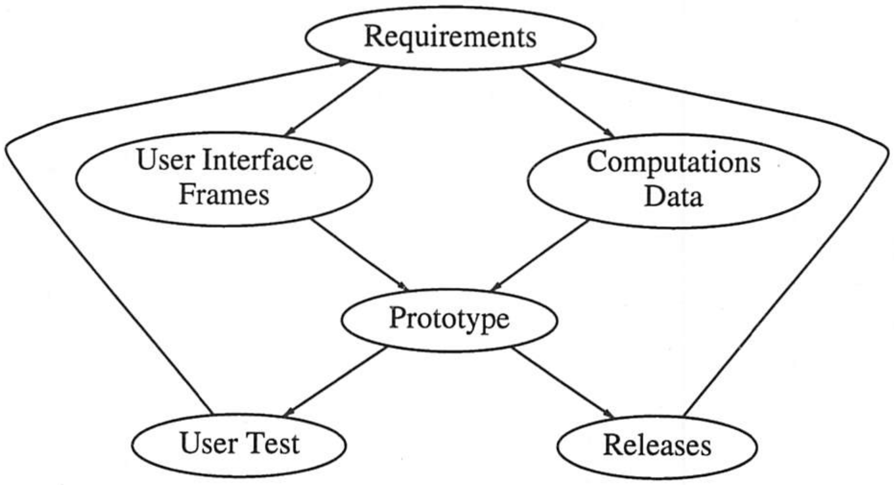


Figure 5: EASEL frame system development cycle

3.2. Building higher abstraction levels with macros

The availability of a language suitable for design programming enables design reuse in the traditional style of building macro templates and code libraries. It is clear how frames parallel traditional library functions and can be reused as such. In many projects where families of applications are built, higher levels of reuse can be achieved by defining macro templates. Such templates are useful because they:

- Reduce programming time of repetitive tasks,
- Standardize the look and feel of the user interface, and

- Standardize the access to lower level computational functions.

For example, consider the task of obtaining a list of logins in frame `users` of the *email* application. The shell communication can be abstracted to the actions of:

- Sending the shell a set of commands,
- Reading responses from the shell, and
- Assigning responses to an appropriate EASEL variable.

Below is a macro definition of this task in an extended C preprocessor language [KR78, Fow88]. The macro `KSH_get` takes two arguments `Cmd` and `Var` that define the command to be sent to the shell and the variable to store any responses. The default values for `Cmd` and `Var` are the string `echo ok` and the variable name `ksh_out`.

```
#macdef KSH_get(Cmd="echo ok", Var=ksh_out)
        {process  "/bin/ksh"    > ~Var
            ~!"ShIsDone\n","echo ShIsDone\n"
            ~$
            Cmd
        }p
#endmac
```

The instantiation below of the macro `KSH_get` generates the equivalent code on lines 2–6 of frame `users`:

```
KSH_get(Var=users, Cmd="ypcat passwd | cut -d: -f1 | sort -u | xargs")
```

The communication protocol with the shell as defined in this example is relatively simple. For other tools, it can be much more complex. By using macro templates, frame developers can focus directly on *what* needs to be performed without having to worry about *how* it is performed, which helps avoid inadvertent errors.

The same technique can also be applied to the menu task in the `select` frame. Below is the macro template to do that. Here, macro parameters are used to describe the main components of a menu block. In this example, the menu window is constrained to always appear with the top left corner at location `x=5` and `y=0` and dimensions `xlen=50` and `ylen=10`. This is somewhat contrived (by default, windows are automatically sized and placed by a best-fit rule) but it serves to show how interface constraints can be standardized using macro templates.

```
#macdef Menu_list( Title=Selections, List=list, Sel=sel, Ans=ans)
      {menu  :(~Sel == ~null)  ~Sel
          .window( x=5, xlen=50, y=0, ylen=10 )
          Title: ~Ans
          {option  ~List  " \t\n"
              ~Ans = ~Ans * ~Sel * " "
          }o
      }m
#endmac
```

For completeness, the following instantiation of `Menu_list` generates the same code for the menu block on lines 2–8 of frame `select`:

```
        Menu_list(Title=Mail To, Sel=pers, Ans=to)
```

4. Experiences with building an application family

Within AT&T, several projects have successfully used EASEL to build sophisticated network management applications. In particular, a project which we shall called Project D made extensive use of EASEL to generate families of end-user applications for a variety of end-users, including:

- Switch engineers – to monitor, forecast, plan, and equip switches to handle traffic loads efficiently,
- Operations support staff – to monitor and evaluate operator services throughout a network, and
- Service planners – to conduct traffic analyses of specialized services and provide reports.

Much of the raw data for these systems is generated automatically by the switches themselves in the telephone network. Portions of this data are periodically downloaded onto UNIX servers and stored in relational databases, for use by applications. These applications are often written in C with embedded SQL calls. The C code can be complex, sophisticated, and not easy to use by non-experts. But to better serve telephone customers, it is becoming more and more important to make such capabilities readily available to end-users.

When Project D first started, it recognized the needs to reuse existing C code and to provide friendly user interfaces. They found lots of literature on user interfaces, e.g., [Sch87], and a variety of user interface packages inspired by the early work at Xerox PARC [Gol84], including Apple Macintoshes and PCs with Microsoft Windows. But what was needed was to wrap a friendly user interface around large existing C applications running on UNIX systems. EASEL was chosen for this task.

In building applications, Project D's architects recognized that there were many tasks that are repeated again and again, such as:

- Obtaining information from the database,
- Formatting data for display to end-users,
- Collecting end-user responses via menus and forms,
- Providing hypertext help,
- Storing new information into the database, and
- Time-stamping and posting messages to end-users.

This led Project D to layer their code as follows:

- A library of high-level objects,
- EASEL frame code,
- Application-specific libraries, including database access utilities (C and SQL), and
- A relational database.

The high level objects currently number about 70 and are implemented as macros that expand into EASEL code (using `awk` and `cpp`). Code generated by the macros reference a library of about 90 utilities, written both in EASEL and C, many of which interface with the database. As new reusable objects or object attributes are found and defined, EASEL code and database access utilities can be expanded as needed.

To date, Project D has built nearly a dozen different end-user systems, each consisting of hundreds of EASEL frames. During the initial phase of the project, many new objects are discovered, implemented in macros and, as necessary, supported by new low-level C functions and/or database transactions. Application architects have said that the capability to write new macros and new code at any level avoids hitting "brick walls" and makes this approach based on EASEL even more powerful than a typical 4GL. Recent applications have been built entirely out of predefined macros. The expansion of common macros can be "personalized" to meet particular requirements of particular customer sets.

A typical Project D source code file written with macros averages about 100 to 200 lines long, which after macro expansion averages about 1,000 lines of EASEL source code. Thus, use of the macros reduces code size roughly by a factor of 5 if compared to straight EASEL code. It is not as easy to get a similar comparison between EASEL code and equivalent code in C but it is not hard to conclude that the code size reduction would be much larger than that. Project D developers have estimated that programming in EASEL alone (without macros) offers a 10 to 1 advantage in terms of implementation time over writing the same end-user system in C.

This high level of software reuse has allowed Project D to employ small teams to interact with customer focus groups, design, test, and rapidly refine prototypes using customer feedback to flesh out features for production releases. The end results for Project D are customers who feel that they are getting what they really want. Finally, from an organizational point of view, an important aspect of having high level macro objects is that they are easily learned by developers new to the project. New Project D developers typically can begin to build screen objects within a few days of training.

5. Evolution and reuse experience

EASEL has grown and matured significantly over the years, thanks to two complementary forces. One has been feedback from designers and developers using EASEL to build end-user systems. Many discovered leading-edge ways to use EASEL, pushing at its frontiers and suggesting new features. The other has been continued involvement and interaction with other software engineering researchers. Discussions frequently led to the recognition of common problems, some of which eventually lent themselves to common solutions to the benefit of more than one project.

An example of users' feedback is the recent addition of an interactive frame builder `efb`. For many years, EASEL provided a language and environment to program and execute end-user systems. We have discussed the benefits of having a language suitable for implementing the high level design of such systems. However, an aspect of user interface design that cannot be easily addressed by a textual language is that of experimenting with different screen layouts. The frame builder `efb` addresses this needs by allowing users to write EASEL code and lay out the screen interactively. It is interesting to note that `efb` itself is implemented in the EASEL language as a frame system consisting of about 150 frames and 3 coprocesses written in C.

EASEL is based on a number of standard libraries; the original list included: *curses*, for screen manipulations, *malloc* for memory allocation, *regex* for regular expression matching [KP84], and *stdio*, the standard input/output library. In the course of EASEL's evolution, we found that some of these libraries needed improvement and that new libraries were needed.

The earliest version of EASEL, around early 1982, was based on the original *curses* library on the BSD4.1 UNIX System. Aside from a number of bugs, this version of *curses* also suffered from lack of features (such as hardware scrolling) and poor performance. After much consideration, *curses* was rewritten to improve code quality and efficiency. The new library, *screen* [FKV94], remains upwards compatible with *curses* and includes new features such as screen editing, menu display, mouse support. It is also fully internationalized and and supports multiple international multi-byte character sets. Because of *screen*, EASEL is perhaps the only current application construction system that enables applications that run transparently in multiple countries using different character code sets across the Orient and Europe.

EASEL uses the *malloc* package for dynamic memory allocations. Early in its development, it was discovered that standard *malloc* implementations both on System V and BSD UNIX systems had severe deficiencies, either in terms of space fragmentation or time performance or both. This prompted a study by Korn and Vo[KV85] to compare all available *malloc* implementations in 1985. At that time, a new *malloc* package based on the best-fit strategy was also implemented. The study found that this package provides the best trade-off in both time and space. This version of *malloc* is now part of the standard System V UNIX distribution. More recently, we found that certain large EASEL applications may have hundreds to thousands of users running concurrently on the same server. This indicates that there is much to be gained by using shared memory for storing frames on line. Coupled with certain other needs, this prompted a generalization of *malloc* to *vmalloc* [Vo94a, Vo94b], a new library that introduces the idea of allocation from regions each of which may employ a different allocation strategy and a different means to obtain memory.

When event handling was added to EASEL, it was discovered that signals could cause the screen to be only partially updated. The bug was tracked down to the *stdio* library which drops data if a `write` system call is interrupted by a signal. This and other shortcomings of *stdio* led to the writing of *sfio*, a faster, safer I/O library [KV91].

Along the way, it was found that several pieces of software in the department, including EASEL, would benefit from a library for on line dictionary management – for example, to access variables in a symbol table. A flexible dictionary library, *libdict* was written to handle both ordered and unordered objects using binary trees or hash tables [NV93].

The need to distribute low-level libraries that depend on system calls and other environmental parameters for a wide range of hardware and software platforms led to the development of an installation tool to automate the process. The tool, called IFFE [FKSV94], probes and then configures the software automatically without human intervention.

5.1. Other approaches

The first version of EASEL called IFS [Vo90] was first developed in the early part of 1982. At that time, there were few alternative languages for building end-user applications. Within AT&T, the best known tool for form-based applications was FE, a form-entry system [Pri85]. As FE focused on forms, it could not be used for more general applications requiring menus or windows. More recently, there are a number of commercial character-based packages with comparable functionality to EASEL. Most notable are the FMLI package distributed with UNIX System V and the JAM package [Jya94]. FMLI is based on the *curses* library and uses a syntax similar to the shell language enhanced with constructs to write forms and menus. The heavy reliance on separate scripts for forms and menus FMLI cumbersome to use. JAM runs on both PC and UNIX systems and is based on proprietary software for screen handling. JAM relies more on an interactive screen builder to prototype screens than on a language to write applications. Though this makes it easy to build single applications with few screens, it can become cumbersome when families of applications must be built along the line of Project D discussed in Section 4. In fact, Project D developers investigated both JAM and EASEL and decided to use EASEL because its open-ended architecture makes it easy to add new functionality at any number of architectural levels, e.g., high-level macros, EASEL's code, processes, or C routines. On the bit-mapped graphic side, the TCL/TK [Ous94] language and toolkit is compatible to EASEL in the basic approach to tool and application design. In fact, it would be interesting to rewrite the display library of EASEL based on TCL/TK or a similar toolkit. An advantage of doing this is that EASEL-based applications could run transparently on character terminals and graphical workstations.

6. Conclusion

We have described EASEL a language and system for building end user systems. From an application builder's point of view, the success of EASEL derives directly from its philosophy of separating *Design programming* from *Computational programming* by providing a programming language suitable for design implementation. This enables a style of system development that focuses on partitioning tasks as seen from users' perspective and increases the chance of matching users' expectations. Having a high level language for design programming brings in traditional reuse techniques such as code libraries and templates. In addition, by defining precisely a small number of standard interfaces to external code, EASEL encourages the construction of reusable code. We described experiences with a project where a family of end-user applications were built based on these ideas. In this case, the application builders themselves claimed at least a factor of five reduction in code size and corresponding productivity improvement.

We have also touched on our own experiences of software reuse in the construction and evolution of EASEL. The total size for EASEL and associated libraries stands at a little over

60,000 lines of C code. For many years, this body of code was maintained and enhanced by essentially a single person (Vo). This is not normally feasible but for the high level of reusability in the internal code. As the software evolves, we gradually abstract pieces of it into reusable libraries. The construction of such libraries have sometimes led to new and interesting theoretical problems. For example, early in the rewrite of the *curses* library, it is recognized that screen scrolling is best modeled by a string matching problem in which matches have weights. This led to the development of a new heaviest common subsequence algorithm[JV92]. Recently, new algorithms and heuristics for memory allocation were developed in building the *vmalloc* library. Thus, reuse permeates our way of building software and drives the interplay between theory and practice.

7. References

[AKW88] Al Aho, Brian Kernighan, and Peter Weinberger. *The AWK Programming Language.* Addison Wesley, 1988.

[Arn84] K. C. R. C. Arnold. *Screen Updating and Cursor Movement Optimization: A Library Package, 4.2 BSD UNIX Programmer's Manual Supplementary Documents.* University of California, Berkeley, July 1984.

[BK89] Morris Bolsky and David G. Korn. *The KornShell Command and Programming Language.* Prentice-Hall Inc., 1989.

[Bou78] S. R. Bourne. The Unix Shell. *AT&T Bell Laboratories Technical Journal,* 57(6):1971–1990, July 1978.

[FKSV94] Glenn S. Fowler, David G. Korn, J. J. Snyder, and Kiem-Phong Vo. Feature-Based Portability. In *VHLL Usenix Symposium on Very High Level Languages,* October 1994.

[FKV94] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for Writing Reusable Libraries. 1994. Available from authors.

[Fow88] Glenn S. Fowler. cpp - The C language preprocessor, 1988. UNIX Man Page.

[Gol84] A. Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addison-Wesley, 1984.

[JV92] Guy Jacobson and Kiem-Phong Vo. Heaviest increasing/common subsequence problems. In *Combinatorial Pattern Matching: Proceedings of the Third Annual Symposium,* volume 644 of *Lecture Notes in Computer Science,* pages 52—65, 1992.

[Jya94] Jyacc. Jam application development guide, 1994.

[KP84] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice Hall, 1984.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Lanugage.* Prentice Hall Software Press, 1978.

[KV85] David G. Korn and Kiem-Phong Vo. In Search of a Better Malloc. In *USENIX 1985 Conference Proceedings*, pages 489–506, 1985.

[KV91] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of Summer USENIX Conference*, pages 235–256. USENIX, 1991.

[NV93] Stephen C. North and Kiem-Phong Vo. Dictionary and Graph Libraries. In *Proceeding of Winter USENIX Conference*, pages 1–11. USENIX, 1993.

[Nye90] Adrian Nye. *Xlib Programming Manual.* O'Reilly & Associates, Inc., 1990.

[Ous94] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[Pri85] Reuben M. Pritchard. FE - A Multi-Interface Form System. *AT&T Technical Journal*, 64(9):2009–2223, November 1985.

[Sch87] Bruce Schneiderman. *Designing the User Interface.* Addison Wesley, 1987.

[Vo85] Kiem-Phong Vo. screen(3X) - more <curses>: the <screen> library, 1985. UNIX Man Page.

[Vo90] Kiem-Phong Vo. IFS: A Tool to Build Application Systems. *IEEE Software*, 7(4):29–36, July 1990.

[Vo94a] Kiem-Phong Vo. Enhancing library usability with disciplines and methods. 1994. Available from the author.

[Vo94b] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. 1994. Available from the author.

Biography

Glenn Fowler is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of NMAKE, a configurable ANSI C preprocessor library, and the *coshell* network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in Electrical Engineering, all from Virginia Tech, Blacksburg Virginia.

John J. Snyder is a Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories, where he has done UNIX systems administration and now works mostly on software for end-user systems. He received a Ph.D. in Econometrics from the University of Colorado in 1979. At that time he worked with FORTRAN on a Cray-1 and UNIX on a DEC PDP 11/70 at the National Center for Atmospheric Research in Boulder. After consulting in Mexico City for a couple of years, he joined AT&T in 1983.

Kiem-Phong Vo is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include aspects of graph theory and discrete algorithms and their applications in reusable and portable software tools. Aside from obscure theoretical works, Phong has worked on a number of popular software tools including the curses and malloc libraries in

UNIX System V, sfio, a safe/fast buffered I/O library and DAG, a program to draw directed graphs. Phong joined Bell Labs in 1981 after receiving a Ph.D. in Mathematics from the University of California at San Diego. He received a Bell Labs Fellow award in 1991.

COMPILING MATLAB

Stephen C. Johnson

Melismatic Software

Cleve Moler

The MathWorks, Inc.

ABSTRACT

MATLAB is a high level language oriented towards scientific and engineering applications. It has evolved over a ten year history to become a popular, flexible, powerful, but still simple language, and has served as an effective platform for over a dozen "toolboxes" supporting everything from symbolic computation to digital filter design, control theory, and neural nets. Designed to be used interactively, MATLAB also supports the ability to define functions and scripts, and dynamically link with C and FORTRAN programs. This paper discusses a project to provide a compiler for MATLAB. The focus of this paper is on techniques that might be useful for other high-level languages, and the parts of such languages that resist compilation. Our constructed MATLAB compiler is in beta test, and has demonstrated speedups over interpretation of up to 300 times on practical programs.

Introduction to MATLAB

MATLAB grew out of an attempt to provide a user-friendly front end for the LINPACK matrix programs developed in the late '70s and early '80s. Visualization methods were added in the late '80s, including a powerful 3-D graphing capability. Recent trends in the language have focused on an object-oriented graphics capability that permits a rich GUI construction, and the support of over a dozen toolboxes that support specialized scientific and engineering areas. A student edition of MATLAB is available for a low price, and over fifty textbooks have been written incorporating MATLAB in their course material.

MATLAB is, for all intents and purposes, a typeless language (or, if you wish, a strongly typed language), since every data object has the same type: a two dimensional array of complex numbers. Vectors are arrays with one dimension equal to 1, while scalars are 1x1 arrays. The sole exception is that strings have a flag bit set to control how they are printed: a string is represented as a vector of real numbers, one byte per element of the array. (This is in amusing contrast to many very very high-level languages that represent numbers as strings; MATLAB represents strings as numbers...). There is also support for sparse matrices, although they intermingle freely with regular matrices. The language is supported by an interpreter that supports interactive use, creation and execution of functions, dynamic loading of FORTRAN and C functions, and extensive debugging facilities.

Syntactically, MATLAB has few surprises. As you might expect, there is a notation for describing two dimensional arrays:

```
        [   23, 2+3i, -5 ;  0, 1, 2 ]
```

describes an array with two rows and three columns. Most MATLAB expressions will print their values when typed--following the expression with a semicolon will suppress the printing. Because many numerical algorithms return multiple values, functions (but not ordinary assignments) can assign to several variables:

```
[L,U] = lu(X);
```

returns the standard L-U decomposition of a matrix X. Functions are also polymorphic:

```
[L,U,p] = lu(X);
```

returns a permutation in addition to the L and U matrices. Functions can interrogate how many inputs and outputs they are called with--in fact, the L matrix returned by lu is different in the two argument and three argument case.

MATLAB users have been encouraged to "vectorize" their computations--that is, to make use of built in array and matrix operations rather than writing scalar code. To this end, a number of vector-like extensions have been put into MATLAB. For example, if I is a vector of 0's and 1's, and A is a vector the same length as I, then A(I) is a vector consisting of just those elements of A where I is 1 (The apparent ambiguity is resolvable (just barely) because subscripts in MATLAB start at 1). Since relational operators and most Boolean operations produce vectors of 0's and 1's, there is no shortage of such index vectors. So you can say A(A>0) to refer to the vector of positive elements from the vector A. You can also delete elements by assigning the empty array to them:

```
A(A<0) = [];
```

deletes all the negative elements from the vector A.

Throughout, the emphasis in MATLAB has been on supporting the user with "no surprises", and the result has been quite successful. However, while the user has few surprises, there are plenty in the implementation. As a simple example, consider the colon operator that produces regularly spaced vectors:

```
0 : .1 : 1
```

produces the vector

```
[ 0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1.0 ]
```

However, in the general case, getting the colon operator to work properly is surprisingly subtle, and quite wound up with the properties of floating-point arithmetic. Simply determining the correct number of elements in the vector is a subtle exercise in roundoff error. For some filtering applications, a colon operator with endpoints $-x$ and x must be exactly symmetrical around the origin, to prevent systematic buildup of roundoff error: contemplating the difficulties with

```
-2*pi : pi/10 : 2*pi
```

where pi is our favorite transcendental will help you appreciate why the colon operator takes several pages of code.

Functions may be defined with a keyword and syntax that gives names to the input and output arguments. A function is written into a file with the extension .m; if the file called xxx.m, and is written into a directory on the search path of the MATLAB interpreter, a function called xxx has just been defined. Note that the function name is derived from the name of the file containing the definition--while the syntax for function declarations allows a name to be specified, this name is ignored. For obvious reasons, these files are called *M-files*.

Mexfiles and the MATLAB Compiler

MATLAB has coexisted with FORTRAN and C code from its earliest days. So pressing was this need that MATLAB has supported dynamic linking and execution for many years, often orginally through special code written before this feature was available as part of standard operating systems. These dynamically loadable "MATLAB executable" files are called *mexfiles*.

An *external interface* was defined to allow Fortran and C programs to be written, compiled, and then dynamically loaded into MATLAB and executed. This interface consists primarily of a set of library functions and an associated header file that allow the Fortran and C programs to determine the dimensions of an array argument, find the real and imaginary parts, construct output arrays, *etc*. Typically, mexfiles consist of a "wrapper" that reads the MATLAB data structures, finds their pieces, and puts together the MATLAB results, and one or more core functions that do the desired computation, often lifted with little change from an existing Fortran or C application.

Mexfiles work like M-files; if a file named `xxx.mex` is on the search path of the interpreter, it is taken to be the definition of the function `xxx`. (Actually, the suffix depends on the system the file is compiled on--for Sun 4's, the actual suffix is `mex4`.). If there is both an `xxx.m` and an `xxx.mex`, the mexfile wins.

The MATLAB compiler compiles M-files into mexfiles, primarily to obtain improved execution speed--when an M-file contains few matrix or vector operations, interpreter overhead can be a considerable fraction of the total runtime, so speedups of orders of magnitude are possible. In writing the compiler, it was a great advantage that there was already an existing dynamically loadable format for mexfiles that could be used. Also, while compiling an M-file to run without the interpreter is sometimes useful (and, in fact, MATLAB supported a toolkit for embedded control that compiled a limited class of M-files into C in a standalone environment), our target for the compiler was to turn M-files into mexfiles, preserving the flexibility and power of the interpeter.

MATLAB's Overhead and How to Remove It

To understand how the compiler speeds up programs, we need first to understand where MATLAB spends its time. The traditional rule of thumb suggests that the "cost" of interpreting is an order of magnitude in execution speed, but this rule of thumb is very misleading when applied to MATLAB. The data objects in MATLAB can be very large, containing thousands of elements. The fundamental operations on these data objects (for example, matrix multiplication) and many of the standard library operations (such as solving linear equations) are built into the interpreter, and use state-of-the-art algorithms. Since these operations can perform millions of floating point operations the overhead of interpretation can be insignificant in these cases.

However, when the MATLAB function consists largely of scalar operations, the overhead can be considerable. Not only do the operations have to be interpreted, but storage needs to be allocated for the matrices, sizes need to be checked, etc. Looking at such a scalar MATLAB program, it appears at first that its translation into C or FORTRAN would be straightforward. For example, consider the simple function that creates an `n`-element vector of squares:

```
function a=squares1(n)
for i=1:n
   a(i) = i*i;
end
```

This is actually very poor MATLAB code, because each vector element assignment causes the array `a` to grow, leading to $n-1$ reallocations of the original space, each time a little larger. Many storage allocators display quadratic behavior with such an allocation pattern, since the previous values must be copied into the larger space for each element of the array. Far better is to allocate a row vector of the right size at the beginning of the loop, and then just stuff the elements:

```
function a=squares2(n)
a=zeros(1,n);
for i=1:n
    a(i) = i*i;
end
```

On a Sun SPARC 10, the second function interprets almost 15 times faster than the first when n is 5000. Not surprisingly, the majority of serious MATLAB functions preallocate such arrays.

However, the really expert MATLAB programmer would just write

```
a = (1:n).^2;
```

which is in turn nearly sixty times faster than `squares2`. (`.^` is an operator that performs the power operation elementwise across the array, calling built in functions).

When the two squares functions are compiled, the generated C code also has to respect the semantics of growing the array with every assignment. The compiled C code for `squares1` runs about 4.5 times faster than the interpreted code for $n = 5000$, due in large part to a somewhat better reallocation scheme for growing the arrays. For `squares2`, the compiled code runs about six times faster than the interpreted code. The C code generated has to check each reference and assignment to ensure that it is in range--a special 'fast' flag allows this subscript checking code (and the resulting semantics of growing the array) to be suppressed, consistent with the spirit of most C code. When this is used, the time is cut by another factor of 6.5, bringing the code quality close to the optimal MATLAB solution. Summarizing the times (in sec.) for n=5000:

```
4.5824  squares1, interpreted
1.0079  squares1, compiled
0.2622  squares2, interpreted
0.0422  squares2, compiled
0.0065  squares2, compiled with 'fast' flag
0.0056  (1:5000).^2
```

Type Imputation

When the `squares2` function is compiled, `i` and `n` are allocated to C integers, and `a` is known to be a real matrix. This saves a great deal of overhead, since the integers are allocated on the stack and no dynamic allocation is needed. Correctly figuring out the "true" types for a MATLAB program is the key part of compiling it effectively. This is especially true when we attempt to determine whether a variable is a scalar, and whether a variable is non-complex. For example, even something as straightforward as `i*i` could take a huge overhead if it was viewed as a matrix multiplication of two complex matrices--hundreds of instructions of overhead would surround the one multiplication that really mattered.

There are three main methods used to gain information about MATLAB types, which we can characterize "bottom up", "top down", and "recursive". Bottom up methods start with a simple known type and follow its possible transformations. For example, in `squares2`, the expression `1:n` is an array of integers, so the `for` statement that defines `i` always sets `i` to an integer. Knowing that `i` is an integer, `i*i` must be, and thus `a(i)` is. Since the elements of `a` are only set to integer values, the array `a` must be an integer array.

The bottom up method is made a bit more complicated because of the dynamic typing of MATLAB. The variable `i` may be the square root of -1 at one place in the program, an integer at another, a string at a third, and a full complex matrix at a fourth use. This means that we cannot have a traditional symbol table that tracks "the" type of `i`; we must associate a type with every use of `i`, and follow the implications of this through the program flow.

An interesting situation arises when we have iterative control flow. For example, suppose the variable `X` is set in three places:

```
X = 1;

for ...
    . . .
    X = X + 1;
    . . .
    X = 2*X;
end
```

It is reasonably clear that `X` is always going to be an integer. However, the `X` on the right side of `X=X+1` has its value set in two places: the initial `X=1`, and the `X=2*X` from the previous loop iteration. Since we know nothing about the type of the previous `X`, we tentatively make `X` an integer. Continuing in this way, we see that `X` always retains an integer value. Had the third equation been `X=2.5*X`, we would have had to upgrade the type of `X` from integer to floating point when we had completed the analysis.

This kind of analysis is well known in compiler writing circles (but not too well known outside), and a well-known technique is used to address this. We analyze the control flow for the program and associate with every variable a list of expressions that might define it (so-called *use-def* chains). When analyzing the type of the `X` on the left side of `X=X+1`, we see that it is set from the right hand side of the assignment, so we need to understand the type of the right-hand X. There are two possible definition points for this X: X=1 and X=2*X. The first definition point has a known type, integer scalar. The second requires us to recursively ask about the type of the left side of X=2*X. This in turn requires that we ask about the right side X of X=2*X. And this has a single definition point, the left side of X=X+1. At this point, we appear to have reached an impasse, but in fact we are nearly done. As we investigate each of these types, we mark each definition of the variable we are recursively examining by setting a flag. When we encounter a definition with the flag set, we know (or, rather, those gifted in such matters can prove) that nothing new will be added from this definition, so we need take no action. At this point, we can unwind the recursion, recognizing that nothing other than integers is ever seen in the type structure for X, so X is an integer everywhere.

This kind of analysis (called a "least fixed point" by the theorists) is easy to apply when the type structure has a natural way of joining two types: for example, joining a floating point number and an integer array gives a floating-point array. We do this analysis with a type hierarchy consisting of integer, float, and complex, and a shape hierarchy consisting of scalar, vector (array with one dimension equal to 1), and array.

The second kind of type imputation arises from top down constraints, e. g., a variable is used in such a way that it implies things about its type. As an example, consider

```
function a=f(x)
    . . .
    y(3) = 4 + x;
    . . .
```

Here, the input variable might *a priori* be any shape, but we know that y(3) is a scalar, so 4 + x must be, so x must be. Unlike the previous case, this type information is not built bottom up from components, but is constrained top down by its uses. As another example, in the squares2 function, we recognize that the arguments to the "zeros" function are integers, so the input n must be.

The challenge comes in correctly blending these two forms of type imputation. For example, suppose we have

A = B * C;

and we know that A is scalar. Then, if we know that B is scalar, we can conclude that C must be scalar. This turns into a top-down requirement on C. If it later develops that we have underestimated the type of B, and B can be a vector, than we must also reexamine C as well.

The third form of type imputation is recursive examination of functions. MATLAB has several hundred library functions in reasonably common use, everything from bookeeping functions like zeros and size to mathematical functions like bessel to decompositions like lu and packages to do entire digital filter and control theory design. These functions are a rich and vital source of type information, and few MATLAB functions go for very long without calling another function, so it is essential not to lose information. Compounding the difficulty is the large number of built in functions.

Because the compiler is viewed as being called from the interpreter the interpreter search path is available, so we can recursively determine the type of all functions called by a given M-file (except for built-ins, that must be kept in a table). To avoid doing this multiple times, we save the computed output types for a given function called with a given number of input and output arguments. For built in functions, we have a table that makes use of the fact that, for most MATLAB functions, the type and shape of the output of a function is related to the type and shape of the inputs. Since most mathematical operations do not cause inputs to become less complicated, the type system we chose is one that relates the types and shapes of the outputs to the types and shapes of the inputs, together with a "lower bound" type and shape.

For example, the absolute value function abs always returns real values, but the shape of the result is the same as the shape of the input. The function prod takes the product of the rows of a matrix or the elements of a vector. The type is the same as the type of the input, but the dimension is one less: prod of a matrix yields a vector, while prod of a vector yields a scalar.

This simple but effective calculus of types is sufficient to handle nearly all built in functions accurately. When a function is used, it is first searched for in the table. If it is not found, the M-file is referenced and the type of the outputs determined recursively. Because the types of the outputs of a function sometimes depend on the number of input and output arguments, the type imputation system makes different table entries for differing numbers of arguments, and can "prune" the parse tree, recognizing conditionals that check the number of incoming or outgoing arguments and eliminating unused code from the function in a particular case.

Wrinkles and Blemishes

As with every language that has been useful and used for a decade, MATLAB has developed some quirks that need careful attention when compiling. For example, in common with many interpreters, MATLAB has an `eval` function that takes a string and evaluates it in the context of the calling function. This causes the compiler so many problems that we just give up on it. To write an `eval` properly we would need to access all local variables and be able to set them. More seriously, `eval` could cause a variable to change type at runtime, totally breaking the assumptions of the compiler.

Unfortunately, there is one common use of `eval` that is reasonably common: using it to simulate a `varargs` functionality in MATLAB. There is some hope that we might recognize this special case and compile it efficiently, but there are also language changes afoot to make this feature a first-class part of the language, so compiling good code will become easy.

Another area where the compiler has difficulty with straightforward code is in dealing with the semantic ambiguity of subscripting. In an expression like

```
A(U) = X;
```

where `U` is an index vector, such vectors may either have values in the range from 1 to the length of `A`, or may have values that are 0 or 1 (as mentioned above), selecting a subvector through a boolean condition. The compiler must, in general, call a function and generate a temporary index vector if needed. In most cases, it should be possible to recognize at compile time which of these cases is present, but this is not currently done in the compiler.

Strings is another area where the compiler struggles. A string datatype was conspicuously missing from the type system described above. Luckily, most MATLAB applications spend little time in string processing. The difficulty comes from many library functions that allow certain arguments to be either strings or values. It can be very difficult to determine at compile time whether such an argument is really a string. This means that we need to carry around a dynamic "string flag" with our matrix structure, as MATLAB does. Given that we need do this anyway, there is little incentive to do a better job recognizing strings at compile time, since string handling rarely accounts for more than a couple of percent of runtime in MATLAB applications.

The operation of removing elements from a vector is another where the compiler needs to be careful in generating code. In MATLAB, an expression like

A(U) = X;

has totally different semantics if `X` is the empty array than if `X` is nonempty. This would *a priori* require the compiler to insert a test and two totally different pieces of code around a large class of such assignments. In practice, virtually 100% of the time when it is desired to remove elements, the matrix `X` is written as explicitly empty. The compiler chooses in this case to restrict the language compiled: we will just not correctly compile the general case when `X` is empty (more precisely, in most cases we will complain dynamically that the dimensions do not agree across the assignment...). This language restriction has caused no problem up to the present.

Unexpected Spinoffs

One interesting result of the compiler project has been a continual flow of bug reports about library routines, some of which have been in constant use for many years. The compiler has found some obsolete but still tolerated usage, some inefficiencies, some failures to check input arguments, and a couple of improper uses (for example, uses of the library routine `size` where `length` was intended).

The compiler is slowly growing a set of *lint*-like features to comment on MATLAB constructions. As an example, consider the expressions

```
    if A == B
```

and

```
    if A ~= B
```

in MATLAB. The first does pretty much what you would expect: the two matrices must agree in shape (or one must be a scalar), and the result of the equality operator is a matrix of 0's and 1's, describing the equality or inequality on an elementwise basis. The semantics of the MATLAB if statment is that the boolean argument is examined, and the positive consequent is taken only if every element of the boolean array is one. So there are no surprises.

The second expression would appear to behave in a similar fashion, but in practice is almost always a bug when `A` or `B` is nonscalar. A boolean array is constructed with 1's if the corresponding elements are unequal, but the if passes only if `all` elements of `A` differ from the corresponding elements of `B`, a decided surprise. Good MATLAB style should really demand one of the library functions `all` or `any` in this case (or, for two dimensional arrays, `all(all(A==B))` and `any(any(A~=B)))`. When the compiler is invoked in verbose mode, users are told about any conditionals that appear to have nontrivial dimension (this also helps the compiler writer find places where the type imputation has missed a trick).

An Example

A prototypical example of the effective use of the MATLAB Compiler is provided by the solution of a tridiagonal system of linear equations. A tridiagonal linear system is n equations in n unknowns where the j-th equation explicitly involves only three unknowns, x_{j-1}, x_j, and x_{j+1}.

$$
\begin{aligned}
b_1x_1 + c_1x_2 &= d_1 \\
a_1x_1 + b_2x_2 + c_2x_3 &= d_2 \\
&\cdots \\
a_{j-1}x_{j-1} + b_jx_j + c_jx_{j+1} &= d_j \\
&\cdots \\
a_{n-1}x_{n-1} + b_nx_n &= d_n
\end{aligned}
$$

The data for the problem consists of two vectors, b and d, of length n, and two vectors, a and c, of length $n-1$. The resulting solution, x, is then a vector of length n.

There are several ways to solve a tridiagonal system in MATLAB. One could create a full, n-by-n matrix, T, with a, b and c on the subdiagonal, diagonal and superdiagonal, and then use the linear system solution operator:

```
    x = T\d
```

As a function of the order, n, this approach requires n^2 storage and n^3 time, so it is impractical for n larger than a few hundred. It is much more efficient to create the sparse form of the same matrix and use the same backslash operator. The time and storage requirements for the sparse linear equation solver are both linear in n, so this approach is reasonable for much larger systems.

Alternatively, one could use the following M-file. This has optimal storage efficiency because it involves only the relevant vectors. But it is slower than the sparse matrix approach because, although the input data and the output result are all vectors, the computation itself cannot be "vectorized". Each iteration of the loops involves only a few scalar operations. (This sample M-file can only be used when the elements of the diagonal coefficient vector, b, are nonzero and, in fact, are large enough that no pivoting for numerical stability is required during the elimination processes.)

```
function x = tridi(a,b,c,d)
%TRIDI  Solve tridiagonal system of equations.
%  x = TRIDI(a,b,c,d) solves the system of linear
%  equations x = T\d where T is the tridiagonal
%  matrix with a on the subdiagonal, b on the diagonal,
%  and c on the superdiagonal, and d is the right hand
%  side.  The input vectors b and d, and the output
%  vector x, all have the same length, say n, and the
%  input vectors a and c have length n-1.

n = length(b);
x = zeros(n,1);

for j = 2:n
    p = a(j-1)/b(j-1);
    b(j) = b(j) - p*c(j-1);
    d(j) = d(j) - p*d(j-1);
end

x(n) = d(n)/b(n);
for j = n-1:-1:1
    x(j) = (d(j)-c(j)*x(j+1))/b(j);
end
```

For this M-file, the compiler's most difficult task is determining that the variables `j` and `n` are integer scalars. This conclusion is possible here because `n` is the output of the length function and `j` is involved in loops which start at integer values and have integer increments. The result is the crucial declaration

```
int j,n;
```

in the resulting C program.

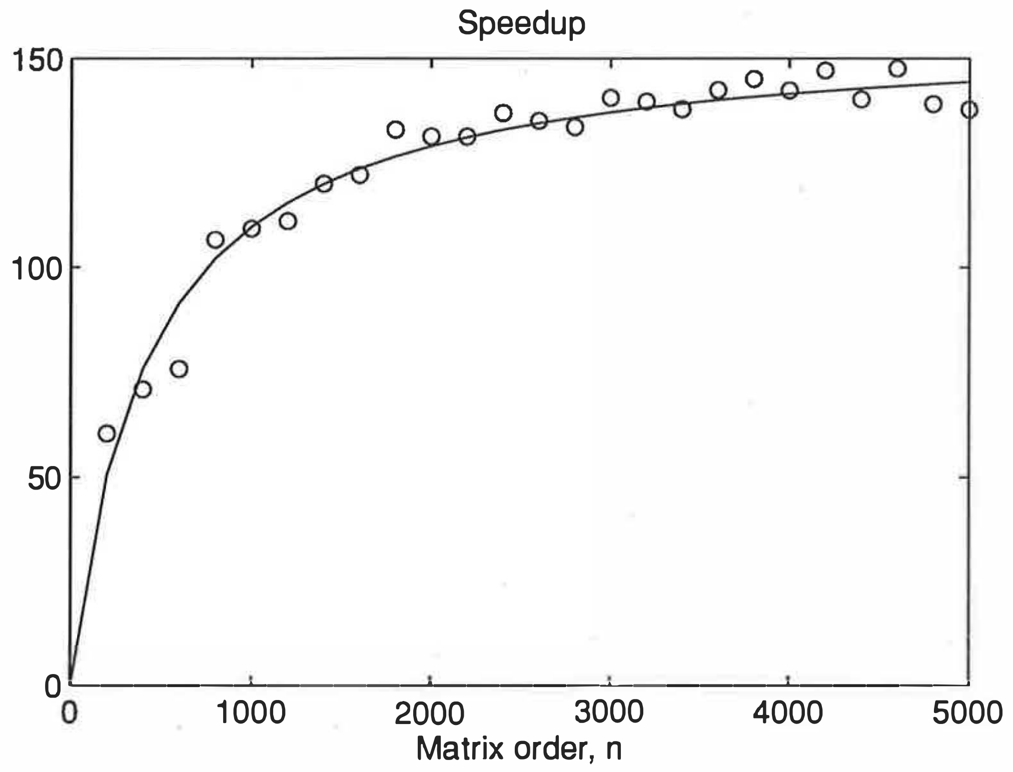
The compiler generates two versions of the body of the function, one for real arithmetic and one for complex arithmetic. Here is the real version.

```
for( j = 2; j <= n; j = j + 1 )
   {
      p = ((a.pr[((j-1)-1)]) / (b.pr[((j-1)-1)]));
      b.pr[(j-1)] = ((b.pr[(j-1)]) - (p * (c.pr[((j-1)-1)])));
      d.pr[(j-1)] = ((d.pr[(j-1)]) - (p * (d.pr[((j-1)-1)])));
   }

   x.pr[(n-1)] = ((d.pr[(n-1)]) / (b.pr[(n-1)]));
   for( j = (n - 1); j >= 1; j = j -1 )
   {
      x.pr[(j-1)] = (((d.pr[(j-1)]) -
          ((c.pr[(j-1)]) * (x.pr[((j+1)-1)]))) / (b.pr[(j-1)]));
   }
```

In the following graph, the circles are the actual measured speedup and the solid line is obtained by fitting the measured execution times with linear functions of *n* and then taking the ratio. We see that, as *n* increases, the compiled version approaches a speedup of 150 over the interpreted version.

Summary

It is very rewarding to write a compiler that speeds programs up by orders of magnitude, especially when the incoming programs are of practical importance. The compiler has not only made MATLAB a more useful product, but has also encouraged an examination of the language and many of the existing programs from a new and interesting perspective.



ksh - An Extensible High Level Language

David G. Korn (dgk@research.att.com)

AT&T Bell Laboratories
Murray Hill, N. J. 07974

ksh is a high level interactive script language that is a superset of the UNIX system shell. ksh has better programming features and better performance. Versions of ksh are distributed with the UNIX system by many vendors; this has created a large and growing user community in many different companies and universities. Applications of up to 25,000 lines have been written in ksh and are in production use.

ksh-93 is the first major revision of ksh in five years. Many of the changes for ksh-93 were made in order to conform to the IEEE POSIX and ISO shell standards. In addition, ksh-93 has many new programming features that vastly extend the power of shell programming. It was revised to meet the needs of a new generation of tools and graphical interfaces. Much of the impetus for ksh-93 was wksh, which allows graphical user interfaces to be written in ksh. ksh-93 includes the functionality of awk, perl, and tcl. Because ksh-93 maintains backward compatibility with earlier versions of ksh, older ksh and Bourne shell scripts should continue to run with ksh-93.

1. INTRODUCTION

One of the earliest very high level languages for the UNIX* system was the shell. The shell, originally an interactive command language, has evolved over the years to become a complete programming language. One of the most popular versions of the shell, ksh, is being used as the primary programming language in many software development projects in many different companies and universities. The most recent version of ksh, ksh-93, incorporates the functionality of awk[1] while maintaining compatibility with earlier shells and with the IEEE POSIX and ISO shell standards.[2]

In this paper I give a history of the UNIX shells, focusing on ksh, primarily the new features in the 1993 version, ksh-93. I also describe the pros and cons of using ksh, rather than awk, perl[3], or tcl[4]. One of the new features of ksh-93 is that it is extensible by applications programmers; I suggest some possible extensions.

To illustrate the programming power of the new version of ksh, I include two shell programs written by Glenn Fowler of AT&T Bell Laboratories. I compare one of these programs with the same program written in perl.

2. HISTORY

The original UNIX system shell was a simple program written by Ken Thompson at Bell Laboratories, as the interface to the new UNIX operating system. It allowed the user to invoke single commands, or to connect commands together by having the output of one command pass through a special file called a pipe and become input for the next command. The Thompson shell was designed as a command interpreter, not a programming language. While one could put a sequence of commands in a file and

* UNIX is a registered trademark licensed exclusively through X/OPEN Corporation

run them, i.e., create a shell script, there was no support for traditional language facilities such as flow control, variables, and functions. When the need for some flow control surfaced, the commands `/bin/if` and `/bin/goto` were created as separate commands. The `/bin/if` command evaluated its first argument and, if true, executed the remainder of the line. The `/bin/goto` command read the script from its standard input, looked for the given label, and set the seek position at that location. When the shell returned from invoking `/bin/goto`, it read the next line from standard input from the location set by `/bin/goto`.

Unlike most earlier systems, the Thompson shell command language was a user-level program that did not have any special privileges. This meant that new shells could be created by any user, which led to a succession of improved shells. In the mid-1970s, John Mashey at Bell Laboratories extended the Thompson shell by adding commands so that it could be used as a primitive programming language. He made commands such as `if` and `goto` built-ins for improved performance, and also added shell variables.

At the same time, Steve Bourne at Bell Laboratories wrote a version of the shell which included programming language techniques. A rich set of structured flow control primitives was part of the language; the shell processed commands by building a parse tree and then evaluating the tree. Because of the rich flow control primitives, there was no need for a `goto` command. Bourne introduced the "here-document" whereby the contents of a file are inserted directly into the script. One of the often overlooked contributions of the Bourne shell is that it helped to eliminate the distinction between programs and shell scripts. Earlier versions of the shell read input from standard input, making it impossible to use shell scripts as part of a pipeline.

By the late 1970s, each of these shells had sizable followings within Bell Laboratories. The two shells were not compatible, leading to a division as to which should become the standard shell. Steve Bourne and John Mashey argued their respective cases at three successive UNIX user group meetings. Between meetings, each enhanced their shell to have the functionality available in the other. A committee was set up to choose a standard shell. It chose the Bourne shell as the standard.

At the time of these so-called ''shell wars'', I worked on a project at Bell Laboratories that needed a form entry system. We decided to build a form interpreter, rather than writing a separate program for each form. Instead of inventing a new script language, we built a form entry system by modifying the Bourne shell, adding built-in commands as necessary. The application was coded as shell scripts. We added a built-in to read form template description files and create shell variables, and a built-in to output shell variables through a form mask. We also added a built-in named `let` to do arithmetic using a small subset of the C language expression syntax. An array facility was added to handle columns of data on the screen. Shell functions were added to make it easier to write modular code, since our shell scripts tended to be larger than most shell scripts at that time. Since the Bourne shell was written in an Algol-like variant of C, we converted our version of it to a more standard K&R version of C. We removed the restriction that disallowed I/O redirection of built-in commands, and added `echo`, `pwd`, and `test` built-in commands for improved performance. Finally, we added a capability to run a command as a coprocess so that the command that processed the user-entered data and accessed the database could be written as a separate process.

At the same time, at the University of California at Berkeley, Bill Joy put together a new shell called the C shell. Like the Mashey shell, it was implemented as a command interpreter, not a programming language. While the C shell contained flow control constructs, shell variables, and an arithmetic facility, its primary contribution was a better command interface. It introduced the idea of a history list and an editing facility, so that users didn't have to retype commands that they had entered incorrectly.

I created the first version of `ksh` soon after I moved to a research position at Bell Laboratories. Starting with the form scripting language, I removed some of the form-specific code, and added useful features from the C shell such as history, aliases, and job control.

In 1982, the UNIX System V shell was converted to K&R C, echo and pwd were made built-in commands, and the ability to define and use shell functions was added. Unfortunately, the System V syntax for function definitions was different from that of ksh. In order to maintain compatibility with the System V shell and preserve backward compatibility, I modified ksh to accept either syntax.

The popular inline editing features (vi and emacs mode) of ksh were created by software developers at Bell Laboratories; the vi line editing mode by Pat Sullivan, and the emacs line editing mode by Mike Veach. Each had independently modified the Bourne shell to add these features, and both were in organizations that wanted to use ksh only if ksh had their respective inline editor. Originally the idea of adding command line editing to ksh was rejected in the hope that line editing would move into the terminal driver. However, when it became clear that this was not likely to happen soon, both line editing modes were integrated into ksh and made optional so that they could be disabled on systems that provided editing as part of the terminal interface.

As more and more software developers at AT&T switched to ksh, it became the de facto standard shell at AT&T. As developers left AT&T to go elsewhere, the demand for ksh led AT&T to make ksh source code available to external customers via the UNIX System Toolchest, an electronic software distribution system. For a one-time fixed cost, any company could buy rights to distribute an unlimited number of ksh binaries. Most UNIX system providers have taken advantage of this and now ship ksh as part of their systems. The wider availability of ksh contributed significantly to its success.

As use of ksh grew, the need for more functionality became apparent. Like the original shell, ksh was first used primarily for setting up processes and handling I/O redirection. Newer uses required more string handling capabilities to reduce the number of process creations. The 1988 version of ksh, the one most widely distributed at the time this is written, extended the pattern matching capability of ksh to be comparable to that of the regular expression matching found in sed and grep.

In spite of its wide availability, ksh source is not in the public domain. This has led to the creation of bash, the "Bourne again shell", by the Free Software Foundation; and pdksh, a public domain version of ksh. Unfortunately, neither is compatible with ksh.

In 1992, the IEEE POSIX 1003.2 and ISO/IEC 9945-2 shell and utilities standards were ratified. These standards describe a shell language that was based on the UNIX System V shell and the 1988 version of ksh. The 1993 version of ksh is a version of ksh which is a superset of the POSIX and ISO/IEC shell standards. With few exceptions, it is backward compatible with the 1988 version of ksh.

The awk command was developed in the late 1970s by Al Aho, Brian Kernighan, and Peter Weinberger of Bell Laboratories as a report generation language. A second-generation awk developed in the early 1980s was a more general-purpose scripting language, but lacked some shell features. It became very common to combine the shell and awk to write script applications. For many applications, this had the disadvantage of being slow because of the time consumed in each invocation of awk. The perl language, developed by Larry Wall in the mid-1980s, is an attempt to combine the capabilities of the shell and awk into a single language. Because perl is freely available and performs better than combined shell and awk, perl has a large user community, primarily at universities.

The need for a reusable scripting language was recognized in the late 1980s by John Ousterhout at the University of California at Berkeley. He invented an extensible scripting language named tcl, an acronym for "tool control language". tcl is written as a library rather than a command. Thus, it can be embedded into any command to give it scripting capability. The tcl language has gained a sizable following, primarily because of an X Window* programming interface that is provided by an adjunct

* X Window is a trademark of Massachusetts Institute of Technology.

tk. With tk it is possible to write X Window applications as tcl scripts.

At about the same time that tk was developed, Steve Pendergrast at UNIX Systems Laboratories created wksh, a program that extends ksh for X Window programming in MOTIF* and OpenLook**. The extensions were added as a collection of built-in commands to create and manipulate widgets. Callback functions are written as shell functions. Another version of ksh for X Window programming similar to wksh, xksh, was developed by Moses Ling at AT&T and is used in several applications. wksh and xksh created new demands on ksh and were a major influence for the new features that have been added to ksh-93. A new windowing desktop shell, dtksh, based on ksh-93 and wksh, has been developed by Novell and will be part of the Common Operating System Environment, COSE.

3. REQUIREMENTS FOR A REUSABLE SCRIPTING LANGUAGE

The primary requirement for any language is that it enable you to easily specify what you want. Also, a scripting language should be able to run without requiring a separate compilation system. Since strings are a basic element of any general-purpose programming language, the language must handle arbitrary length strings automatically.

A general-purpose scripting language should be simple to learn. There is no easy way to measure how simple a language is to learn, but the time to learn a language can be reduced by making the language similar to one that users already know. For instance, arithmetic computations should use familiar notation and operators should have conventional precedences.

The language should be widely available and well documented in order to achieve wide usage. Programmers do not want to spend time learning a language that will not be available to them wherever they work. Also, since no single document is right for everyone, there should be several documents for different types of users.

In addition to performing arithmetic, a script language should have string and pattern matching capabilities. The details of memory management of variable sized objects should be handled by the language, not by the user. Many applications require the handling of aggregate objects. Even though very high level languages require fewer lines of code, real world applications are likely to be large; thus a good script language needs to have a method to write procedures or functions that have automatic variables and that can return arbitrary values.

Applications coded in the language should have performance comparable to that of the application if it were written in a lower level language. This means that the overhead for interpretation must be amortized by the useful work of the application. The lower the overhead for interpretation, the larger the class of applications for which the language will be useful. Some applications are short-lived and will be dominated by the time they take to start up.

For many applications, the language should interface simply with the operating system. It should be possible to open or create files and network connections, and to read and write data to these objects. It must be possible to extend the language in application-specific domains to achieve high reuse.

Finally, the language should make it easy to write portable applications. One should be able to write scripts that do not depend on the underlying operating system, file system, or locale.

* MOTIF is a registered trademark of Open Software Foundation, Inc.

** OpenLook is a registered trademark of UNIX System Laboratories, Inc.

4. HOW `ksh` IS USED

The most frequent use of `ksh` is as an interactive command interpreter. In this context, most users learn the basics of redirection and pipelines. The most important features for this use are command line editing and history interaction.

A second common use of `ksh` is for writing scripts that combine several commands into a single command. These scripts are usually placed into the user's private *bin* directory. These scripts are customized to the individual's needs and usually are not designed for reuse. However, it is not uncommon for some of these scripts to be useful to a broader community of users. In this case, the script can be moved into a public *bin* directory that is accessible by a larger group of users.

A third common use of `ksh` is as an embedded scripting language. In addition to library calls `popen()` and `system()`, tools such as `make` and `cron` have used the shell as their specification language. One weakness of the traditional C-to-shell interface is that the shell actions are carried out by a separate process so that no side effects are possible. With `ksh-93` it is possible to have the shell interpreter run in the same process.

A fourth use of `ksh` is for writing administrative scripts. Since all UNIX systems are delivered with scripts written in the shell language, system administrators need to be able to read and write scripts to do their job. The most important features for these applications are the ability to generate and test files, and the ability to invoke pipelines.

A fifth use of `ksh` is for the generation of front ends. `ksh` provides a coprocess mechanism which makes it easy to run a process that is connected to a shell script via pipes. The script interacts with a user, and then generates commands to send to the coprocess to carry out most of the work. With `wksh` and `dtksh`, the front ends can be graphical.

A sixth use of `ksh` is for program generation. Scripts can be written that produce code for compiled languages such as C and C++, or for script languages such as `ksh`. For this use, the ability to handle arbitrary strings and patterns is essential. The ''here-document'' feature is well suited for program templates. The `iffe` command described elsewhere in this proceedings [5] uses the shell in this way.

The final use of `ksh` is for writing programs. In this context, `ksh` does virtually all the work without relying heavily on other utilities. This is the area in which shells have traditionally been weakest and the reason that languages such as `perl` and `tcl` were invented. The new version of `ksh`, `ksh-93`, eliminates this weakness.

5. NEW FEATURES IN `ksh-93`

`ksh-93` is the first major revision of `ksh` in five years. It was revised to meet the needs of a new generation of tools and graphical interfaces. Much of the impetus for `ksh-93` was `wksh`, which allows graphical user interfaces to be written in `ksh` as `tk` allows one to write graphical user interfaces in `tcl`. The intent was to provide most of the `awk` functionality as part of `ksh`, as does `perl`. Because `ksh-93` maintains backward compatibility with earlier versions of `ksh`, older `ksh` and UNIX System V shell scripts should continue to run with `ksh-93`.

Many of the changes for `ksh-93` were needed in order to conform to the POSIX shell standard. In addition, `ksh-93` has many new programming features that vastly extend the power of shell programming.

In this section, several important new features introduced in `ksh-93` are described.

5.1 Floating Point Arithmetic

Applications that required floating point arithmetic no longer have to invoke a command such as `awk` or `bc`. The comma operator, the **?:** operator, and the pre- and post-increment operators were added. Thus `ksh-93` can evaluate virtually all ANSI C arithmetic expressions. An arithmetic `for` command, nearly identical to the `for` statement in C, was added. In addition, the functions from the math library were added.

5.2 Associative Arrays

Earlier versions of `ksh` had one-dimensional indexed arrays. The subscripts for an indexed array are arithmetic expressions which are evaluated to compute the subscript index. Associative arrays have the same syntax as indexed arrays, but the subscripts are strings; they are useful for creating associative tables. However, because the list of indices is not easily determined, a shell expansion character was added to give the list of indices for an array. Because associative arrays reuse the same syntax as indexed arrays, it is easy to modify scripts that use indexed arrays to use associative arrays.

5.3 Additional String Processing Capabilities

Shell patterns in `ksh-93` are far more extensive than in the Bourne shell, having the full power of extended regular expressions found in `awk`, `perl`, and `tcl`. In addition, `ksh-93` has new expansion operators for substring generation and pattern replacement. Substring operations can be applied to aggregate objects such as arrays.

5.4 Hierarchical Name Space for Shell Variables

One of the lessons learned from the UNIX system is that a hierarchical name space is better than a flat name space. With `ksh-93`, the separator for levels of the hierarchy is **.** (dot). It is possible to create compound data elements (data structures) in `ksh-93`. Name references were added to make it easier to write shell functions that take the name of a shell variable as an argument, rather than its value. With earlier versions of `ksh`, it was frequently necessary to use `eval` inside a function that took the name of a variable as an argument.

Shell variables in `ksh-93` have also been generalized so that they can behave as active objects rather than as simple storage cells. This has been done by allowing a set of discipline functions to be associated with each variable. A discipline function is defined like any other function, except that the name for a discipline function is formed by using the variable name, followed by a **.** (dot), followed by the discipline name. Each variable can have discipline functions defined that are invoked when the variable is referenced or assigned a new value. This allows variables to be active rather than passive. For example, by defining the discipline function

```
function date.get
{
    date=$(date)
}
```

the value of `$date` will be the output of the `date` program. At the C library level, variables can be created that allow for any number of discipline functions to be stacked and associated with a variable. These functions can be invoked in an application-specific way. For example, an X Window extension can associate each widget with a shell variable, and the user can write callback functions as discipline functions.

5.5 Formatted Output

One of the most annoying aspects of shell programming is that the behavior of the `echo` command differs on various systems. The lack of agreement of the behavior of `echo` in the POSIX standard led to the requirement for `printf`. In `ksh-93`, `printf` is a built-in command that conforms to the ANSI C standard definition and has a few extensions. The two most important extensions are the `%P` format conversion which treats the argument to be converted as a regular expression, and converts it to a shell pattern; and the `%q` format conversion which prints the argument quoted so that it can be input to `ksh` as a literal string. These two simple extensions make it much easier to correctly write scripts that generate scripts.

5.6 Runtime Built-in Commands

With `ksh-93`, a user can add built-in commands at runtime on systems that support dynamic linking of code. Built-in commands have much less overhead to invoke, and unless they produce side effects, they are indistinguishable from commands that are not built in. Each built-in command uses the same argument signature as `main()`, and returns a value which is its exit status. Complete applications can be written in `ksh-93` by writing a library that gets loaded into `ksh-93` for the application-specific portion. The C language shell interface allows the user to access the shell variable name space and to insert C language discipline functions for variables.

To give an example of the performance improvement that arises by having a command built in, the script

```
for ((i=0; i < 1000; i++))
do   cat
done < /dev/null
```

takes approximately 20 seconds to run on a Silicon Graphics workstation when `cat` is not a built-in command, and takes .2 seconds to run when `cat` is a built-in.

The ability to extend the shell at runtime makes many new uses of `ksh` possible. Here are examples of some possible extensions:

5.6.1 Writing Servers One possible extension would make it simple to write servers in `ksh`. The addition of `/dev/tcp`... and `/dev/udp`... with redirection, already allows clients to be written as `ksh` scripts. A built-in command could be added to advertise the service and to associate it with a variable. Callback functions that handle message events could be written as discipline functions for this variable. A second built-in command would then process events received by the server and invoke the appropriate discipline function.

5.6.2 Persistence A built-in command can be added which declares that a portion of the variable name space of a script be persistent. The built-in would take a second argument that maps this store onto the file system. Each assignment to a variable under this part of the variable name space would also cause the file system to be updated.

5.6.3 Object-oriented Database Manipulation Built-in commands can be added to read and write objects stored in an object-oriented database. The objects can be represented as shell variables, and the methods as shell discipline functions.

5.7 Support for Internationalization

The earlier version of `ksh` was eight-bit transparent and had a compile option to handle multibyte character sets. The behavior of `ksh-93` is determined by the locale. In addition to the earlier support for internationalization, `ksh-93` handles:

- Character classes for pattern matching. In `ksh-93` one can specify matching for all alphabetic characters in a locale-independent way.
- Character equivalence classes for pattern matching. In `ksh-93` one can specify matching for all characters that have the same primary collation weight as defined by the locale definition file.
- Collation. The locale you are in determines the order in which files and strings are sorted.
- String translation. One can designate strings to be looked up in a locale-specific dictionary at run time by preceding the string with a $; for example, `$"hello world"`.

- Decimal point. The character that represents the decimal point for floating point numbers is determined by the current locale.

5.8 Usability As a Library

ksh-93 has been rewritten so that it can be used as a library and called from within a C program. This makes it possible to add ksh-compatible scripting capabilities to any command, similar to the way one can with tcl.

6. ADVANTAGES OF USING ksh

6.1 Advantages of ksh Compatibility with Bourne Shell

Compatibility with the Bourne shell reduces the learning curve and makes it possible to reuse the many thousands of existing scripts. Because of the large number of Bourne shell users, there is a large community who already know how to use ksh.

Because ksh is compatible with the Bourne shell, there is no limit to the length of variable names, the length of strings, or the number of items in a list. File manipulation and pipeline creation are simple, and "here-documents" allow script applications to be packaged into a single file.

Compatibility with the Bourne shell makes ksh a better interactive language than most other high level scripting languages. Having the same language for interactive use as for programming has several advantages. First of all, it reduces the learning curve since everything learned for interactive use can be used in programs and vice versa. Secondly, interactive debugging is simpler since it uses the same language as do the programs.

6.2 Advantages of ksh Over the Bourne Shell and POSIX Shell

Using ksh rather than the Bourne shell has many additional advantages other than improved performance. The inline editing feature makes ksh friendlier to interact with. Since the inline editing feature can be enabled by scripts that are read from the terminal, interactively debugging ksh scripts is easier.

The Bourne shell is notorious for its lack of arithmetic facility. The expr command is both slow and awkward. The string processing capabilities are inferior to languages that are based on regular expressions. ksh uses ANSI C style arithmetic, and a pattern matching notation that is equivalent to regular expressions.

ksh has a better function mechanism than the Bourne shell or POSIX shell. Large applications are difficult to write with the Bourne shell or POSIX shell because of an inadequate function facility. Bourne shell and POSIX shell functions do not allow local variables. In addition to allowing local variables, ksh allows functions to be linked into an application when they are first referenced, making it possible to write reusable function libraries. ksh is also more suitable for larger applications because it has better debugging facilities.

One advantage of using ksh is that it has been around for several years and has a large user community. The result is that ksh is well documented. There are several books on ksh.[6] [7] [8] [9]

6.3 Disadvantages of Compatibility with Bourne Shell

There are some drawbacks to using a Bourne shell compatible shell as a programming language, many of which have been rectified in ksh. One drawback, the unfortunate choices in the quoting rules in the Bourne shell, makes it more difficult to write scripts that correctly handle strings with special characters in them. tcl is better in this respect since it uses a different character to begin a quoted string than to end the string, which makes nested quoting much easier. The use in ksh of the `$(...)` syntax in place of `` ` ` `` is a major improvement that allows easy nesting of command substitution. Another Bourne shell mistake that remains is that ANSI C sequences are not expanded inside double quoted strings.

This makes it hard to enter non-printable characters in a script without sacrificing readability. It also leads to the different behaviors of the `echo` command on different systems. With `ksh-93`, any single quoted string preceded by $ is processed using the ANSI C string rules.

A second problem with using a Bourne shell compatible language is that field splitting and file name generation are done on every command word. In purely string processing applications, this is not the desired default, thus these operations are better left to functions as with `perl` and `tcl`. With `ksh` it is possible to disable field splitting and/or file name generation on a per function basis, which makes it possible to eliminate this common source of errors.

A third drawback is that scripts depend on all the programs they invoke, and these programs may not behave the same on all systems. In addition, there are variations in the versions of the shell that exist on different systems. To overcome this, `ksh` has more built-in capabilities so that scripts can be less dependent on system commands.

A fourth drawback is performance. On many UNIX systems, the time to invoke a non built-in command is about 100 times more than the time to run a built-in command. This means that, to achieve good performance, it is necessary to minimize the number of processes that a script creates. To overcome this problem, `ksh` has much more built-in functionality so that more operations can be performed without creating a separate process.

7. A PROGRAMMING EXAMPLE

I have chosen a real world example to illustrate the programming power of `ksh`. This example was chosen because the programs are small enough to be included in their entirety and yet are in production use. One of the programs has also been coded in `perl` so that direct comparisons can be made. The `perl` code is in the **APPENDIX**. However, understanding the programs requires some knowledge of the `make` program or a similar software configuration program.

The programs exhibited below were written by Glenn Fowler of AT&T Bell Laboratories to deal with the porting of makefiles to different systems. As is commonly known, the standard UNIX system `make` program is inadequate for large project development. As a result, several other `make` programs have been written and used in software development. At AT&T, the most frequently used make tool is `nmake`[10], written by Glenn Fowler. Typical `nmake` makefiles are about an order of magnitude smaller than old `make` makefiles, they keep track of more information, and they are portable across virtually all UNIX systems. However, to build software on machines that do not have `nmake`, it is necessary to translate the `nmake` makefile to a format that can run on the given machine.

An intermediate `make` language called MAM, which stands for *Make Abstract Machine*, has been developed by Glenn Fowler. Using an intermediate form to express the information in a makefile reduces the number of such translation programs necessary to migrate from any `make` variant to another. A make abstract file, called a *mamfile,* encompasses the static and dynamic nature of `make`. It has a simple instruction set that has been used to abstract both old `make` (System V, BSD, GNU) and `nmake`. `make` abstractions form the basis for makefile conversion tools, makefile porting, regression testing, and makefile analysis. Tools written to process mamfiles can be used with any version of `make`.

Generating a mamfile from most versions of `make` requires a relatively straightforward source code modification. The code modification for GNU `make` has been forwarded to the Free Software Foundation.

7.1 The MAM Language

To understand the `ksh-93` programs below, I give a brief description of the MAM language. The MAM language provides a simple, concise notation for describing `make` entities (variables, rules, actions) and relationships (dependencies).

The mamfile syntax is akin to assembly. It consists of a sequence of instructions, one per line, read from top to bottom. The rule and variable definitions instructions are as follows:

`make` *rule* [*attribute* ...]

`done` *rule* [*attribute* ...]

> A `make/done` pair defines the target rule named *rule*. Nested `make/done` pairs define the dependencies: the enclosing *rule* is the parent, and the enclosed *rules* are the prerequisites. The optional *attributes* classify the rule or dependency relationship (older `make` programs may not emit any attributes) such as `archive`, `implicit`, and `virtual`.

`prev` *rule* [*attribute* ...]

> Used to reference rules that have already been defined by `make/done`.

`exec` *rule* [*action-line*]

> Appends *action-line* to the action (or recipe) for *rule*. The *rule* name – (minus) names the rule in the current `make/done` nesting. `exec` is not emitted until all prerequisite rules for *rule* have been defined.

`setv` *variable* [*value*]

> Assigns *value* to *variable*. A `setv` is required for each variable referenced in the mamfile, even if its value is null, and it must occur before the first reference of the variable. This allows an analysis program to determine all variables used in the makefile and the proper assignment order.

7.2 MAM Example

The makefile in Exhibit 1 illustrates MAM. The corresponding mamfile is listed in Exhibit 2, annotated with line numbers for easy reference. The first column contains mamfile line numbers and the second column contains line numbers from the makefile in Exhibit 1.

Exhibit 1. Sample makefile

```
1    DEBUG = -g -DDEBUG=1
2    CCFLAGS = $(DEBUG)
3    cmd : cmd.o lib.o
4        $(CC) $(CCFLAGS) -o cmd cmd.o lib.o
5    cmd.o : cmd.h lib.h
6    lib.o : lib.h
```

The `info mam` instruction (line 01) identifies the file as a mamfile, lists the MAM version, and also identifies the program that generated the mamfile. Variables are defined using `setv` (lines 02-04). Lines 02 and 03 come directly from the makefile and line 04 is inferred from the predefined rules. `exec` defines the rule actions (lines 13, 19, 21), but is not emitted until after all prerequisite rules have been defined. The rule name – (minus) is shorthand for the rule name in the current `make/done` nesting (lines 13, 19, 21).

Makefile variable references are converted to shell syntax in the mamfile (lines 03, 13, 19, 21). This provides a common target language for actions, and also makes it easy to analyze makefiles using the shell.

7.3 Programs to Process Mamfiles

The first program named `mamold`, listed in Exhibit 3 below, is a program that converts a `mamfile` to a makefile that can be processed by old versions of `make`. This program attests to both the power of `ksh-93` and the simplicity of MAM. A `perl` version of this program is included in the **APPENDIX**.

Exhibit 2. Resulting mamfile

```
01   -   info mam 01/01/94 oldmake
02   1   setv DEBUG -g -DDEBUG=1
03   2   setv CCFLAGS $DEBUG
04   -   setv CC cc
05   3   make cmd
06   3   make cmd.o
07   -   make cmd.c
08   -   done cmd.c
09   5   make cmd.h
10   5   done cmd.h
11   5   make lib.h
12   5   done lib.h
13   -   exec - $CC $CCFLAGS -c cmd.c
14   3   done cmd.o
15   3   make lib.o
16   -   make lib.c
17   -   done lib.c
18   6   prev lib.h
19   -   exec - $CC $CCFLAGS -c lib.c
20   6   done lib.o
21   4   exec - $CC $CCFLAGS -o cmd cmd.o lib.o
22   2   done cmd
```

The original `mamold` program was written as an 800-line C program about five years ago. Since it generates old `make` makefiles, `mamold` must differentiate between explicit (`prereqs[`*rule*`]`) and implicit (`implicit[`*rule*`]`) prerequisites for a given *rule* (lines 22-24). `setv` (lines 18-19) converts special *mam* variables in to shell syntax and prints the name and value. The assertions are emitted in mamfile order (lines 27, 39-44), and the `closure` function emits the transitive closure of the explicit and implicit prerequisites.

This example uses several of the features of `ksh-93` that were not in earlier versions of `ksh`. Line 26 uses the pre-increment operator (the post-decrement operator is used on line 34), which is also new in `ksh-93`. Line 18 uses the new string global replacement parameter expansion operator to change *mam* variables from shell syntax to `make` syntax.

Line 12 defines associate array variables `prereqs`, `implicit`, and `action` that are used elsewhere in the script. Lines 31, 32 and 42 use the new ANSI C string notation to represent newlines and tabs in a readable fashion. The remainder of the script is compatible with earlier versions of `ksh`, but uses several features that are not in the Bourne shell or the POSIX shell. In particular it uses the `[[...]]` compound command for pattern matching and the `((...))` command for arithmetic.

The performance of the `mamold` shell script has been measured against both the original C version and the `perl` script. The `nmake` makefile for a library was converted to a mamfile. The 136-line `nmake` makefile produced a 2912-line mamfile. The mamfile was converted to a 1510-line old `make` makefile using each of the `mamold` programs. The results for user+sys times in seconds on an unloaded SPARC 2 are presented in Exhibit 4.

In this example, the C version was more than an order of magnitude longer and more than an order of magnitude faster than the `ksh` or `perl` version. The `ksh` version is about half the size and about 5%

Exhibit 3. mamold: MAM to old make makefile converter

```
set -o noglob  #disable file expansioin
# generate implicit+explicit in list
function closure {
      typeset i j
      for i
      do    [[ " $list " == *" $i "* ]] && continue
            list="$list $i"
            for j in ${implicit[$i]}
            do closure $j; done
      done
}
typeset -A prereqs implicit action
typeset -i level=0
typeset rule list order target
print "# # makefile generated by mamold.sh # #"
while read -r op arg val
do    case $op in
      setv) [[ $val == *'${mam_'* ]] && val=${val//'${mam_'/'$${mam_'}
            print -r -- $arg = $val
            ;;
      make|prev) rule=${target[level]}
            [[ " $val " == *" implicit "* ]] &&
            implicit[$rule]="${implicit[$rule]} $arg" ||
            prereqs[$rule]="${prereqs[$rule]} $arg"
            [[ $op == prev ]] && continue
            target[++level]=$arg
            [[ " $order " != *" $arg "* ]] && order="$order $arg"
            ;;
      exec) [[ $arg == - ]] && arg=${target[level]}
            [[ ${action[$arg]} ]] &&
            action[$arg]=${action[$arg]}'$(newline)\'$'\n'$'\t'$val ||
            action[$arg]=$'\t'$val
            ;;
      done) ((level--))
            ;;
      esac
done
# dump the assertions
for rule in $order
do    [[ ! ${prereqs[$rule]} && ! ${action[$rule]} ]] && continue
      list=
      closure ${prereqs[$rule]} && print -r -- $'\n'"$rule :$list"
      [[ ${action[$rule]} ]] && print -r -- "${action[$rule]}"
done
```

slower than the perl version.

A second example is the program mamexec, listed in Exhibit 5. A program about four times the size of this has been written in standard Version 7 UNIX Bourne shell as well. The mamexec program executes a mamfile the way make executes a makefile. Using mamfiles and mamexec provides a

Exhibit 4. comparative mamold times (seconds)

	user	sys	user+sys
C	0.48	0.35	0.83
ksh	18.28	0.43	18.71
perl	16.83	0.74	17.57

way to port applications to environments that have the shell, but do not have nmake.

The mamexec source is straightforward. Lines 01-13 initialize the local variables (lines 01-02), associative arrays (line 03), and options (lines 04-13). Lines 15-16 compare the current state with the previous state (if it exists). same[*rule*] is 1 if *rule's* current modification time is the same as the statefile modification time (ls determines the time, comm and sort determine entries that have not changed). A rule is out-of-date if same[rule] is null. The main loop (lines 17-38) collects prerequisites and actions, and executes actions for out-of-date rules (line 33). Only built-in commands are executed in the main loop. setv assigns variables that have no previous value, unless the setv occurs inside a make/done nesting (line 20). Lines 39-40 update the statefile before the script exits. The state consists of two files; mamfile.ml contains the list of all rules, and mamfile.ms contains the state time and name for all rules, where *mamfile* is the name of the mamfile.

This program illustrates some other new ksh-93 features. The getopts built-in command on line 04 is an extension of the earlier getopt version. It allows option strings inside [and] to be inserted as part of the getopts option list. These strings are ignored when searching for options, but are used to automatically generate usage messages. It also uses associative arrays and the ANSI C string syntax. Line 43 uses the name parameter expansion operator ! to generate the list of subscripts for the associative array list. This program illustrates how natural it is to integrate traditional shell constructs such as pipelines and command substitution with string processing code.

Exhibit 6 is a table of user+sys time in seconds on an unloaded SPARC 2 for various make -n commands running on a makefile (or mamfile) with 97 up-to-date rules and 223 action lines. Since mamexec is a ksh script and the other make programs are compiled programs, one might assume that mamexec would perform poorly, but this is not the case. There are a few reasons for this: the mamfile parse is simple compared to makefiles, and all rule bindings are precomputed in the mamfile (i.e., metarules and file path searches have already been applied by the mamfile generator). nmake is slightly slower in this example because it computes the implicit prerequisites that have already been asserted in the mamfile and old make makefiles, and the startup cost for this outweighs the small makefile size.

8. CONCLUSION

ksh has proven to be a good choice as a scripting language. It has the capabilities of perl and tcl, yet it is backward compatible with the Bourne shell. Applications of up to 25,000 lines have been written in ksh and are in production use.

As stated earlier, a language needs to be widely available and well documented to achieve wide usage. The previous version of ksh is shipped with most UNIX systems, and ksh-93 will be on COSE systems. ksh-93 is currently being licensed by AT&T under the name pksh. Plans are underway to make ksh-93 source code available through the UNIX system Toolchest.

Based on a few simple tests, the performance of ksh is comparable to that of perl. While the performance for the mamold in this paper was about 5% worse with ksh than with perl, another applications written in both perl and ksh-93, have shown the reverse.* Limited testing with tcl has shown similar results.

Finally, because ksh is extensible, new reusable components are likely to be implemented as ksh libraries. Extensions for graphical user interface programming provided by dtksh are likely to be widely available. Additionally, it will be possible to use ksh with tk for graphical user interface programming.

* The application was a purify to gprof converter

Exhibit 5. `mamexec:` `make` with state

```
typeset beg="(set -ex;" end=") </dev/null" exec=eval mamfile=Mamfile
typeset -i accept=0 force=0 level=0
typeset -A list same
while getopts "AFf:[mamfile]n" opt
do      case ${opt#${opt%?}} in
        A) accept=1 ;;
        F) force=1 ;;
        f) mamfile=$OPTARG ;;
        n) exec=print beg= end= ;;
        esac
done
[[ $opt == "?" ]] && exit 1
ml=$mamfile.ml ms=$mamfile.ms
[[ $force == 0 && -f $ml && -f $ms ]] &&
for i in $(ls -ld $(<$ml) 2>/dev/null|sort|comm -12 $ms -|sed 's/.* //')
do same[$i]=1; done
while read -r op arg data
do      case $op in
        setv) eval val='$'$arg
                [[ $level != 0 || $val ]] && eval $arg='$data </dev/null'
                ;;
        make) level=level+1
                eval arg=$arg
                [[ " $data " == *" metarule "* ]] && same[$arg]=1 && continue
                name[level]=$arg cmds[level]= list[$arg]=1
                ;;
        prev) eval arg=$arg
                [[ ! ${same[$arg]} ]] && same[${name[level]}]=
                ;;
        exec) [[ ! ${cmds[level]} ]] && cmds[level]=$data ||
                cmds[level]=${cmds[level]}$'\n'$data
                ;;
        done) eval arg=$arg
                [[ ! ${same[$arg]} ]] &&
                {   [[ ${cmds[level]} ]] &&
                    { $exec "$beg${cmds[level]}$end" || exit; }
                    same[${name[level-1]}]=; }
                level=level-1
                ;;
        esac
done <$mamfile
[[ $accept == 1 || $exec == eval ]] &&
{ print ${!list[@]} > $ml; ls -ld ${!list[@]} 2>/dev/null|sort > $ms; }
```

Exhibit 6. `comparative make -n times (seconds)`

	user	sys	user+sys
gnumake	0.35	0.30	0.65
oldmake	0.43	1.08	1.51
mamexec	1.23	0.40	1.65
nmake	0.85	0.95	1.80

REFERENCES

1. Al Aho, Brian Kernighan, and Peter Weinberger, *The AWK Programming Language,* Addison Wesley, 1988.
2. *POSIX – Part 2: Shell and Utilities,* IEEE Std 1003.2-1992, ISO/IEC 9945-2, IEEE, 1993.
3. Larry Wall and Randal Schwartz, *Programming perl,* O'Reilly & Associates, 1990.
4. John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994, as a very high level scripting language.
5. Glenn S. Fowler, David G. Korn, John J. Snyder, and Kiem-Phong Vo, *Feature Based Portability*, Proceedings of the USENIX Symposium on Very High Level Languages, 1994.
6. Morris Bolsky and David Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1989.
7. Morris Bolsky and David Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.
8. Anatole Olczak, *The Korn Shell - User & Programming Manual*, Addison Wesley, 1991.
9. Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly & Associates, Inc., 1993.
10. G. S. Fowler, *The Fourth Generation Make*, USENIX Portland 1985 Summer Conference Proceedings, pp. 159-174, 1985.

BIOGRAPHY

David Korn received a B.S. in Mathematics in 1965 from Rensselaer Polytechnic Institute and a Ph.D. in Mathematics from the Courant Institute at New York University in 1969 where he worked as a research scientist in the field of transonic aerodynamics until joining Bell Laboratories in September 1976. He was a visiting Professor of computer science at New York University for the 1980-81 academic year and worked on the ULTRA-computer project (a project to design a massively parallel super-computer).

He is currently a supervisor of research at Murray Hill, New Jersey. His primary assignment is to explore new directions in software development techniques that improve programming productivity. His best know effort in this area is the Korn shell, `ksh`, which is a Bourne compatible UNIX shell with many features added. The language is described in a book which he co-authored with Morris Bolsky. In 1987, he received a Bell Labs fellow award.

APPENDIX

```
#!/usr/local/bin/perl

# convert MAM to oldmake makefile

print "# # makefile generated by mamold.pl # #\n";
$level = 0;
while (<>) {
        chop;
        ($op, $arg, @val) = split;
        if ($op eq "setv") {
                if ("@val" =~ /\$\{mam_/) {
                        local($buf) = "@val";
                        $buf =~ s/\$\{mam_/\$\$\{mam_/g;
                        @val = $buf;
                }
                print "$arg = @val\n";
        }
        elsif ($op eq "make" || $op eq "prev") {
                $rule = $target[$level];
                if ("@val" =~ /\bimplicit\b/) {
                        $implicit{$rule} .= " $arg";
                }
                else {
                        $prereqs{$rule} .= " $arg";
                }
                if ($op eq "make") {
                        $target[++$level] = $arg;
                        order: {
                                foreach $i (@order) {
                                        if ($i eq $arg) {
                                                last order;
                                        }
                                }
                                $order[$#order+1] = $arg;
                        }
                }
        }
        elsif ($op eq "exec") {
                if ($arg eq "-") {
                        $arg = $target[$level];
                        if ($action{$arg}) {
                                $action{$arg} .= "\$(newline)\\\n\t@val";
                        }
                        else {
                                $action{$arg} = "\t@val";
                        }
                }
        }
        elsif ($op eq "done") {
                $level = $level - 1;
        }
}

# dump the assertions

for ($i = 0; $i <= $#order; $i++) {
        $rule = $order[$i];
        if ($prereqs{$rule} || $action{$rule}) {
                @list = ();
                %mark = ();
                &closure(split(/ /,$prereqs{$rule}));
                print "\n$rule :@list\n";
                if ($action{$rule}) {
                        print "$action{$rule}\n";
                }
        }
}

# generate explicit+implicit into @list

sub closure {
        local($i);
        if ($#_ >= 0) {
                foreach $i (@_) {
                        if ($mark{$i} != 1) {
                                $mark{$i} = 1;
                                $list[$#list+1] = $i;
                        }
                        &closure(split(/ /,$implicit{$i}));
                }
        }
}
```

Fornax: A General-Purpose Programming Language

J. Storrs Hall
Laboratory for Computer Science Research
Rutgers University
`josh@cs.rutgers.edu`

Abstract

Fornax[1] is a very high-level programming based on pattern-matching concepts from Snobol (and Icon) and array or "data-parallel" concepts from *APL* (and J).

Introduction

During the 1980's it was the practice of the Student Chapter of the ACM at Rutgers to hold annual programming contests. This author had the honor of being one of the judges at such a contest. It worked as follows: The entrants were teams, of from 3 to 5 students. Each team was given an assignment of four programs to write. The team which got all four programs running correctly the earliest, won.

Normally most of the students used Pascal, which was the language taught in computer science courses, or Basic, which many had started in. A few engineering students did their programming in Fortran. This year there was entered a motley team of mavericks. Instead of working together, each member would do his work alone, and in a different language. One, who had interned at Bell Labs, would work in C. Another, God help him, was going to use assembly language. A third would try Lisp. And the other member of the "screwball" team, whose name was Damian Osisek, worked in SNOBOL.

The contest was unexciting that night, because Mr. Osisek, working alone, completed all four programs before anyone else, individual or team, completed even one.[2]

Philosophy

The general reaction to this phenomenon, and similar feats occasionally seen in *APL*, is to dismiss them as not applicable to larger, multi-person projects. We believe the truth is not so simplistic. If there is a tool, such as SNOBOL, which can rapidly solve problems of size x, it is the job of the software project manager

[1] Fornax, "the Furnace," is an obscure Southern Hemisphere constellation between Cetus and Eridanus.

[2] The following year, the use of SNOBOL (and *APL*) was banned.

to reduce the project to problems of size x and combine the results. In fact, the manager must do this anyway, and tends naturally to prefer software systems that alleviate the integration task; he is less concerned about the actual programming of the subparts because other people do that.

The other major objection to SNOBOL (and *APL*) is that they are interpreted, and thus unsuited for large systems that are heavily used or where performance is critical. This objection carries forward to the more powerful of the modern languages such as Prolog and the functional languages. It is perhaps even more cogent in the latter case, since Prolog is notoriously difficult to control; runtimes are not only slow, but unpredictable.

The design of Fornax is based on the notion that the state of the art in compilers has advanced sufficiently since the mid-1960's when SNOBOL and *APL* were invented, that reasonably efficient implementations of their primitives can now be compiled. The design goals of Fornax are, in order of importance:

- o A set of datatypes and data-manipulation primitives that are powerful, well-matched to each other, but which can be implemented efficiently.
- o A concise, uncluttered syntax. The austere beauty of Miranda is not to be achieved, but we can avoid the garbage-dump appearance of Common Lisp.

The primitive operations, of course, are part of an abstract world that includes the ontology of data-structures and the framework in which the operations are done. Logic and functional languages attempt to abstract away from the notion of a small sequence of changes to an overall state; lower-level conventional languages overindulge in it as being an efficiently implementable reflection of the actual process in hardware. Fornax includes state changes to be used where necessary, and indeed encourages this as quite a powerful primitive concept, but also provides primitives for operations that are conceptually parallel or unitary to avoid gratuitous decomposition.

Elements

The overall structure of a Fornax program is a set of function definitions, as in Lisp, *APL*, or the various logic and functional languages. Like Prolog, there may be several entries with the same function name, which are selected between on the basis of pattern matching with the arguments. Unlike Prolog, there is no backtracking. There are vector operators like *APL* and string pattern matching like SNOBOL.

Within a given definition the syntax is a simple operator expression. Undeclared variables are lexically scoped and local to the function definition. Side-effects are allowed but restricted to the action of explicit assignment operators; parameter passing is by value.

There are two kinds of expressions, pattern and value. A pattern can be thought of as a program that takes a string as input and either accepts (matches) it or not. Pattern matching has many of the features of programs, e.g. variable assignment can be done as part of a match. The left-hand sides of assignment statements, and the formal parameters of functions, can be patterns, with interesting and sometimes useful results.

Arrays

The datastructure in Fornax is a recursive, multidimensional array. It is similar in semantics to the "array theory" array of Trenchard More. Arrays may be "closed" to create synthetic scalar objects; objects of arbitrary complexity may be created in this fashion and treated as atomic by other code. This capability is intended for for such "object oriented" uses; multidimensionality is sufficient for more pedestrian aggregations.

While no concrete datastructure implements Fornax arrays efficiently for all operations, linked lists, indexed memory arrays, trees, hash tables, and so forth are appropriate for various subsets. It is intended that the programmer ignore the details and use the abstraction as needed, and the compiler will determine how best to implement the resulting operations (even changing data representations if necessary).

An array in Fornax code is written as a sequence of values, e.g. `6 7 8`. A "sequence" of one value, is an array of one value, is a scalar, e.g. `4`. There is no requirement that each subarray of an array have the same number of elements, or indeed have the same dimensionality. The "nested" and "multidimensional" interpretations of array structures in Fornax are unified; any array can be treated either way at any time.

Scalar Extension and Parallel Application

In *APL*, `2 + 3 4` automatically extends the `2` to the length of the vector `3 4` and automatically does + in parallel between the resulting vectors, producing `5 6`. Fornax does not do this; most of the functions operate on more complex structures than numbers, and it would be quite ad hoc to "destructure" them to some arbitrary level to do the parallelization automatically.

Instead parallelization is specified explicitly, using the "`.`" operator. `2 +. 3 4` produces the vectorization we want. In general, `a f. b` produces a result that is the same length as `b`, of which each element is `a f [the corresponding element of b]`. Likewise `a .f b` produces a result corresponding to `a`.

Extension is often used in conjunction with literal functions; any expression in `{}` is taken as a function; `&` and `&&` represent its right, and if supplied, left arguments. To take the absolute value of each number in `list` (without using `abs`), do `{&>0|-&}. list`.

Success and Failure

As in SNOBOL and Prolog, expressions in Fornax can succeed, returning a value, or fail. Normally any expression having a subexpression which fails, will itself fail; some constructs, used for control flow, "catch" failure. Unlike Prolog, backtracking is not done.

Arrays can contain failures as elements. Any "dotted" function application catches single-element failures.

L-values and R-values

In conventional programming languages, only variables, subscript expressions, and record field expressions have L-values, that is, only those expressions that refer to an explicit storage cell. SNOBOL allows the result of a pattern match to be an L-value, doing replacement of the substring that matched. As special cases, insertion and deletion could be done, replacing a null string with a non-null one and vice versa. Fornax adopts this formulation of L-values, allowing a fairly general range of expressions.

Patterns can be used as L-values, generally as the formal parameters to functions. Variable names in this context are interpreted as equivalent to `__`, the pattern that matches any object. Thus we can define list reverse as:

```
rev (a,,b) := (,b), rev a
rev () := ()
```

(`b` is rotated so it must match a single element; `a` can match any length of list, including 0, preceding the last element.)

Patterns

Pattern matching happens in two modes. The first is exact matching, denoted by `^`, which is essentially an equality predicate. The other mode is a substring match, denoted by `?`, where the pattern must match some substring of the subject array. That is to say, `s ? p` if there are arrays `a`, `b`, and `c` such that `s = a,b,c` and `b^p`.

The simplest patterns are values. A value, as a pattern, matches itself and nothing else: `5 6 ^ 5 6` and `4 5 6 7 ? 5 6` succeed, but `6 5^ 5 6` and `3 4 5 ? 5 6` fail. The L-value, as well as the R-value, of a successful `^` is its left argument. A `?` returns an array of the substrings matched; its R-value is the sequence of places in the string where the matches occurred. If

```
a = "peter piper picked a peck of pickled peppers"
```

then `a ? "pe"` returns `"pe" "pe" "pe" "pe" "pe"`, and `a ? "pe" .= ""` leaves `a` equal to `"ter pir picked a ck of pickled prs"`.

Fornax has a set of pattern primitives allowing the construction of patterns roughly equivalent to the ones in SNOBOL. Pattern functions and value functions are overloaded onto the same symbol set, sometimes subtly: `v1, v2` is the concatenation of `v1` and `v2`, whereas `p1, p2` is a pattern that matches any concatenation of something matching `p1` and something matching `p2`. On the other hand, `p1 | p2` is the pattern alternation of `p1` and `p2` (just as in SNOBOL) but `v1 | v2` is a flow-of-control expression like a Lisp OR.

All value functions and their pattern cognates have the same precedence; an expression parses the same whether it is in value or pattern context.

Examples

Suppose we want to count the characters in a file, say standard input. Although normally the line-at-a-time association is used for standard input, direct association is usually used for other files, and can be used for standard i/o if desired. The associated variable is stdio. Directly associated variables have the file's contents as a value; changing the variable changes the file. When interactivity is not necessary, this is usually a more straightforward approach to most algorithms:

```
    flen = # stdio
```

Now suppose we want to count words instead of characters. The easiest way to do that is pattern matching. If we had a pattern that matched a word, we'd be well on our way to counting them. Fornax has built-in patterns alf and dig that match alphabetic characters or digits. Unary ? creates a pattern:

```
    word = ? (+ digit) | (+ alf), ["'", + alf]
```

Now for the pattern matching function, binary ?. It takes a string (actually, any array) on the left and a pattern on the right, and returns a list of all the places the pattern matched in the list; but only non-overlapping ones. When possible matches overlap, the one that starts earliest wins, and if two start at the same place, the longer one wins. That's just what we need; the length of the list of matches is the number of words:

```
    nwds = # stdio ? word
```

Since Fornax actually has the pattern word built in, as well as one called line, the Unix utility `wc` can be written:

```
    tty = (# stdio ? line) (# stdio ? word) (# stdio)
```

Alternatively, one could write:

```
    tty =  #. stdio ?. line word _
```

where _ is the primitive pattern that matches atomic objects, characters in this case.

This example shows how the results of a pattern match are used as data in further operations. More complex results can be produced by explicitly specifying the "value of the match" in the pattern. `p^x` is a pattern that matches whatever pattern `p` would match, but instead of returning the string that matched as its value, the match returns the value of `x`. (Any occurence of `&` in `x` has the value of the string that `p` matched.)

The following pattern definitions, assuming `var` and `num` are defined, match an arithmetic expression and produce as a result a Lisp-like nested prefix parse tree for it. A similar pattern could evaluate the expression in the process of parsing it:

```
factor := ? var | num | "(", (?exp), ")"
term := ? factor | x:factor, "*", y:term ^ 'times' x y
            | x:factor, "/", y:term ^ 'quo' x y
exp := ? term | x:term, "+", y:exp ^ 'plus' x y
            | x:term, "-", y:exp ^ 'sub' x y
```

then `"4+x*3-y" ^ exp` would produce

```
'plus 4 'sub 'times x 3' y''
```

assuming `var` and `num` produced symbols, and noting that `'...'` is a syntax for quoted nestable symbol lists.

Note also that the monadic question mark in the definition of `factor` prevents the parentheses being returned as part of its value. `"(", (?exp), ")"` is equivalent to `"(", x:exp, ")" ^ x`, see below.

Conclusion

Fornax was originally designed as a general-purpose implementation language for an experimental associative architecture with hardware parallel and pattern-matching primitives (the Rutgers CAM). However, we feel it has significant contributions to make on conventional architectures, given the state of the art in compiler technology.

Fornax's current implementation status is "under construction." Our strategy is a dumb interpreter and a smart compiler, in Fornax, to be bootstrapped. Fornax, as would be expected from its antecedents in SNOBOL, is a marvelous language for writing compilers. We find that Fornax programs are typically 25 times shorter than their equivalents in raw C.

A Abridged Dictionary of Fornax

The following is an incomplete list of Fornax functions, operators, and constants. Note that many of the function symbols have both value and pattern cognates; the precedences of cognates is always the same so that parsing is consistent. Precedences are arranged so that a few stereotypical statements, e.g.

```
(vec ? ( (((v:p),(v:p))^exp)
       | (((v:p),(v:p))^exp) ))
= ((x^y) ! exp)
```

can be written without parentheses.

Value Expression Functions

`()` (*Nil*) A list with no elements. It is identical to `''` (no symbols) and `""` (no characters).

`: n` (*Index Vector*) `n` must be a non-negative integer; the result is a vector of length `n` consisting of the integers `0` through `n-1`. This is one form of an

idempotent array, i.e. one in which translates each valid index value into itself.
Often the argument to `:` is `_`, the primitive atomic pattern. Generally this is when the result is involved in a paired operation (e.g. `.+.`) with another vector. Primitive patterns may be used this way when some lexically apparent value (i.e. the length of the other array) will alone allow the expression to succeed.

`x = v` (*Assignment*) The L-value `x` is assigned the R-value `v`. This function has lower than standard precedence and returns the previous value of `x`; it is intended as the root operation of statement-level expressions.

`u f= v` (*Modification*) Has the effect of doing `u = u f v`, except that `u` is evaluated only once. `:=` is interpreted idiosyncratically as function or constant definition.

`x : v` (*Assignment*) The L-value `x` is assigned the R-value `v`. This function has higher than standard precedence and returns the value assigned; it is intended for use within other expressions.

`f / v` (*Reduction*) The elements of `v` are `f`'ed together from left to right; e.g. `+/1 2 3` is 6. Failure elements act as failures in expressions in a reduction. Thus `&/a` returns the final element of `a` if there are no fail elements, and `|/a` returns the first non-fail element.
The two-argument form `u f/ v` is equivalent to `f/ (,u),v`.

`n / v` (*Drop*) `v` with the first `n` elements removed. As in *APL*, negative values of `n` remove elements from the end.

`f\ v` (*Scan*) A parallel-prefix operator. Unlike *APL*, the definition of scan is based on a left-to-right accumulation, (rather than a sequence of partial reductions).

`u \ v` (*Take*) `u` is a non-negative integer. The result is a vector of the first `u` elements of `v`.
Take and drop can be used to specify substrings, either as L-values or R-values. If `a = "themselves"`, then `2\3/a` is `"ms"`; and after the evaluation of `2\3/a = " "`, `a` is `"the elves"`.

`~ v` (*Not*) Fails if `v` succeeds. Returns `()` if `v` fails.

`u ~ v` (*Not Equal*) Compares the values `u` and `v`, or if one or both are patterns, pattern matches between them. If the match succeeds, it fails. Otherwise returns `u`.

`? p` (*Pattern*) If found in a value context, evaluates `p` as a pattern.

`v ? p` (*Pattern Match*) Pattern matching: substrings (in the first dimension) of `v` that match `p` are sought. The value (L or R) is a vector of the matching substrings, left to right, maximal, and non-overlapping.

```
"aaabbc" ? x:_, *x
```

returns `"aaa" "bb" "c"`, even though the pattern actually matches 10 substrings of the original.

& (*Argument*) In {}, & is niladic (takes no arguments) represents the right argument to the function. E.g. `{&-2}. 3 8 4` is `1 6 2`.
In the value part, v, of a `p^v` expression in a pattern, & represents the portion of the string matched by the p part.

`u f|g v` (*Functional Or*) Equivalent to `u f v | u g v`, but only evaluates u and v once.

`u | v` (*Or*) If u succeeds, its value is returned and v is not executed. Otherwise, the value of v. (The so-called Cambridge OR, from its use as a control structure in LISP. The corresponding Cambridge AND is implicit in the actions of failure and sequencing. An if-then-else-like construct could be written `(if; then) | else`.

`^ v` (*Return*) Returns v as the value of the current function, defined or {}, or, if in a []-protected expression, returns v as the value of the expression. An exact if-then-else construct could be written `[(if; ^then) | ^else]`.

`u ^ v` (*Equals*) Compares the values u and v, or if one or both are patterns, pattern matches between them. If the match succeeds, returns the match value, or fails if it fails. The L-value of a successful match is the left argument.

`@ v` (*Downrotate*) The opposite of uprotate (,). Has no effect on scalars. `@()` fails. To `process` each element of a list sequentially, do

```
@list ! process @(1\list = ())
```

(in practice, one would probably do `process. list`).

`u @ v` (*Indexing*) u is an array, and v is an index, e.g. `9 8 7@0` is 9, `(2 3) (4 5)@1 0` is 4, and so forth. Out-of-range indexes fail. Items selected are automatically downrotated.

`$ v` (*Indices Of*) The indices of v (along the first dimension), as a vector. Equivalent to `:#v` unless v has missing or non-numeric indices.

`u $ v` (*Associate*) A vector which maps the elements of u as indices, to corresponding elements of v. Can create symbolic indices.

`# v` (*Length*) The number of elements along the first dimension of v.

`n # v` (*Dup*) n copies of v concatenated together.

`* v` (*Iterate into List*) v is evaluated iteratively until it fails. The result is a list of the values produced. If the first attempt fails, the result is an empty list.

`+ v` (*Iterate into non-null List*) v is evaluated iteratively until it fails. The result is a list of the values produced. If the first attempt fails, the expression fails.

`u + v` (*Addition*) Similarly, - is subtraction, * is multiplication, is division.

`- v` (*Negation*) Of numbers.

`< v` (*Close*) Returns an atomic scalar object that can be opened to produce v. Closing lists of characters produces symbols.

u < v (*Comparison*) Less than; > is greater than. Both comparison operators, as well as ^, return the left argument if they succeed, making constructions such as `a<b<c` work properly. `<|^/list` tells you if a list is sorted. All Fornax objects fall into a global collating order; `1 < 1 1 < 1 2 < 2 < 2 1 < 2 2`.

> v (*Open*) Returns whatever was closed to produce v.

, v (*Uprotate*) In the sense of changing the direction of the dimensional axes of v. The primitive function `rot` does a cyclic shift on the elements along the first dimension. Note that rotating a scalar (in either sense) does nothing.

u , v (*Concatenate*) Along the first, i.e. outermost, i.e. top-level dimension.

! v (*While*) v is iterated until it fails. Returns an integer, the number of times the iteration succeeded.

u ! v (*While*) v is iterated until u fails. Returns an integer, the number of times the iteration succeeded.

; v (*Identity*) Returns v.

u ; v (*Sequencing*) Returns v. Side effects of u and v are done in that order. Has very low precedence; intended to separate statement-level expressions.

Pattern Functions

_ and __ (*Primitive patterns*) _ matches any scalar, and __ matches anything.

x : q (*Assign at Match*) In a pattern, the left argument of : a variable name which is assigned the substring (or substructure) which matches the pattern that is the right argument. The value is valid lexically throughout the pattern, even before the assignment. (The idea is to put the assignment in the most easily matched position, for efficiency, or equivalently, allow the compiler latitude to do the assignment from any reference. E.g.

```
(1 2 3 4) 2 ? (a:__, b, c:__) (b:_)
```

assigns `a = 1`, `b = 2`, and `c = 3 4`.)

˜ p (*Pattern Negation*) As a pattern, matches any string s for which `s ? p` would fail, i.e. anything not containing a match for p as a substring.

```
ccomment := ? "/*", (˜ "*/"), "*/"
```

p ˜ q (*Pattern Difference*) Matches anything that matches p and not q. Not the same as `p & ˜ q`.
Note that pattern negation and difference are somewhat more intuitive methods of obtaining the same pattern semantics provided by SNOBOL's "fence" and Prolog's "cut".

? p (*Selected Subpattern*) The value as a pattern is the same as the argument; but when the whole pattern matches, the value of the (overall) match is the part that matched the argument of the monadic ?. Instead of writing

!p (break; q.v.) one could write (?*_),p. Note, however, that the scope of ? is the lexical pattern it appears in.

p ? q (*Unification*) Matches anything that matches both p and q, i.e. the resulting pattern is the most general unifier of the two patterns.

| p (*Optional*) Matches p or a null string. May also be written [p].

p | q (*Alternation*) Pattern disjunction, i.e. matches anything that matches p or q.

^ n (*Pos*) In a pattern, matches a null string at position n (0 is left end).

p ^ v (*Pattern Value*) In a pattern, matches whatever p matches but the "value of the match" is v. "&" in v means whatever substring p actually matched.

* p (*Arbno*) Matches any number of p's, including none.

+ p (*Nonempty Arbno*) Matches at least one p and as many more as available.

n \ p (*Take*) n is a non-negative integer. Matches any list of length n which is an initial substring of a list which would match p. Drop (\) works similarly. There are subtle differences in the value and pattern forms; 1\() as a value, fails, but as a pattern, will match a gap in the subject list.

! p (*Break*) A generalization of the function in SNOBOL. Matches everything up to, but not including, any substring that would match p.

p ! q (*Any Order*) Matches either p,q or q,p. However, if either argument is itself a ! expression, any permutation of the constituent arguments is allowed.

Graphics Programming in Icon Version 9

Clinton L. Jeffery
Division of MCSS, The University of Texas, San Antonio, TX 78249, U.S.A.
E-mail: jeffery@ringer.cs.utsa.edu

Ralph E. Griswold and Gregg M. Townsend
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.
E-mail: {ralph|gmt}@cs.arizona.edu

Abstract

Version 9 of the Icon programming language introduces support for graphics and user interface programming as an extension of the existing file-based input/output model, rather than introducing graphics as a disjoint facility. Simplicity, generality, and portability are the primary emphases. The result is a language in which common graphic effects are easy to write using ordinary procedural code. Complex techniques such as event-driven programming are optional and not forced on the programmer when they are not needed.

Background and Objectives

Support for graphics and interactive programming usually consists of complex libraries that are difficult to learn by virtue of their girth. Libraries and toolkits found in many languages often make supported behavior trivial at the cost of making unsupported behavior difficult or impossible; many toolkits also emphasize the "user interface" aspect of graphical user interface programming while neglecting the goal of providing flexible graphics output capabilities. Interface construction tools help, but they too have limitations.

In the Icon programming language, we have added support for graphics and interface programming directly to the language. For the sake of those maintaining existing programs, as well as clarity and ease of learning, Icon's graphics facilities focus on simplicity and integration of older file-based and newer interactive input/output models.

Icon's graphics facilities consist of 47 functions and an extension of an existing type. Experience has shown that new users can write windowing applications immediately, yet the facilities are complete enough for development of sophisticated interfaces. Resulting programs run unmodified on X Window, and OS/2 Presentation Manager; MS Windows, Windows NT, and Macintosh support are pending. Ease of programming may be exemplified (if only simply) by Brad Myers' challenge issued prior to the CHI '92 conference: how hard is it to write a program in which the user uses a mouse to move a rectangle around on-screen?

Another important consideration in the language design was the integration of the graphics subsystem into the existing language. This issue addresses the difficulty with which the new input/output facilities are adopted in existing programs and by existing programmers. A question we might use to evaluate a design is: how hard is it to add mouse support to existing text-oriented applications?

An objective underlying these issues was to leave the programmer in control of the program. Most graphics facilities mandate *event-driven programming*. The basic tenet of event-driven programming is that at every instant the user, instead of the program, should be in charge of the interaction with the computer. When taken too far in language design, like most dogmas, the benefits of event-driven programming are achieved at the cost of an enforced program complexity that often is not necessary. In particular, it makes the job of adding graphics or the use of a mouse in text-oriented applications more difficult than is necessary. In Icon the programmer is free to decide when and how much the event-driven programming paradigm should be used. Programmers can write programs in the same way they always have. Adding interface features such as mouse input to a text-oriented application represents a minor enhancement rather than a redesign of program logic.

These three design criteria—simplification, integration, and control—determine the general characteristics of Icon's graphics facilities, and result in several beneficial side-effects. For example, the simplicity of the graphics facilities results in increased ease of implementation; there are fewer functions and features that must be implemented in order to make Icon run on a new window system.

Overview of the Graphics Facilities

Icon's graphics facilities provide a new data type, window, and operations on that new data type. There are several fundamental ways in which this type and its operations simplify the large number of types and functions needed for graphics programming and window management in most systems. This section presents key features of Icon's graphics facilities. See [Jeff94] for a complete description.

Windows as terminals

Type window is an extension of Icon's file data type. Windows may be substituted for files in the language's existing file input/output operations. In such usages, a window operates in a manner similar to a computer terminal. A window has a more substantial internal state than most terminals, including a text cursor position (analogous to conventional terminal cursor position), the current window contents, as well as various font, color, and graphics style attributes that affect the appearance of output.

In addition to the window-as-terminal programming model, every window simultaneously supports graphics operations on a two-dimensional array of pixels. There is no mode-switching between text and graphics. Graphics input/output does not affect the text model in any way, and vice versa, except that output in either model overwrites and obscures prior output in the same location within the window. Viewing windows as "graphics terminals" is consistent with the model actually provided by window systems such as MGR [Uhle88] and $8\frac{1}{2}$ [Pike91].

There is no concept of window exposure in Icon; window repainting is handled automatically by the language. Retained windows are essential in providing the programmer with the freedom to organize program control flow in a manner that is appropriate to the application instead of requiring that organization revolve around window system events.

Encapsulation of complex features in windows

A call to an Icon function typically results in many underlying graphic system calls. Similarly, the underlying system objects used during window input/output operations, such as network connections, graphics attributes, and graphics context information are all packaged together into a single source-level window value. The programmer may ignore them and can expect reasonable default behavior. This approach makes simple applications easy to develop and allows gradual increases in sophistication and functionality as an application matures.

Encapsulating multiple system objects and composing higher-level operations from multiple system calls are techniques typical of higher-level toolkits and languages. Icon can be viewed as an extreme case of the use of these techniques: Window system independence and ease of learning are achieved by implementing all window system operations in terms of normal language values such as strings and integers.

Graphics

The graphics facilities provide the kind of generality and flexibility found in the rest of the Icon language. Icon's graphics functions all take an optional window parameter followed by an arbitrary number of arguments, and automatically convert arguments to the appropriate type (such as integer pixel coordinates). Special types for graphical entities such as points and rectangles are not employed. Many arguments may be omitted and default to values appropriate for the graphics function in question. For example, a circle is obtained by drawing an arc at a specific location and with a specific width; the height of the arc defaults to the width and the starting angle and extent of the arc default to produce a complete circle. This kind of defaulting behavior provides a concise programming notation while minimizing the number of functions the programmer must learn. A further example is provided by the use of the default window, &window. This global variable provides a default window argument to the various graphics drawing functions. The net result is that in order to draw a circle, the Icon programmer typically need only write

```
DrawArc(x, y, diameter)
```

Because this simplification is achieved by the provision of default values, it does not reduce the capability of the function repertoire; it merely allows simple operations to be specified in a simple way and in a notation that is consistent with related but more complex operations.

Window Attributes

An Icon window consists of a hierarchy of underlying window system components. For manipulation of these components Icon abstracts a complex set of internal structures into a single set of fifty six *attributes* and associated *values*. Attributes and values are queried and assigned by means of the function WAttrib(). For example, the call

```
WAttrib("height")
```

returns the height of &window in pixels. The attribute may be changed by following it with an equal sign and a value:

```
WAttrib("height=300")
```

No language is perfect; the attribute model has proven to be effective but the functional notation provided by WAttrib() is cumbersome. There are various shorthand notations in Icon for commonly-used attributes and combinations of attributes; a record-style window.attribute notation would improve the attribute mechanism in a more general way. In any case, the basic simplicity of the attribute-value model provides a conceptual framework that is independent of the window system and is easily learned.

Canvases and Contexts

Although graphics systems vary widely in their programming interfaces, the abstractions underlying various attributes fall into two categories: *canvas* attributes describe the physical appearance of the window, such as its size and position on the screen, while *context* attributes affect the behavior of various operations, such as the colors used in drawing.

The most important advantage of distinguishing canvas from context is to allow multiple contexts, each with varying attribute values, to be used on a given window without the necessity to save and restore relevant attributes. The function Clone() provides this capability in Icon; Clone(w) returns a window value whose operations apply to the same canvas as its window argument w, but whose context is a distinct copy of the argument's context. Collections of clones are routinely used to provide direct "color by number" mappings from application-domain values to window values with varying color or font settings.

Color and Font Naming

Colors and fonts pose some of the most serious portability issues in the Icon graphics facilities, because the variation in color capabilities and font support among platforms is large. The problem of variation in color naming is addressed by a standard set of color names defined by the language. Names such as "light blue" are converted into numeric RGB values which are passed to the system. The color names are inspired by a system proposed earlier [Berk82], with certain additions.

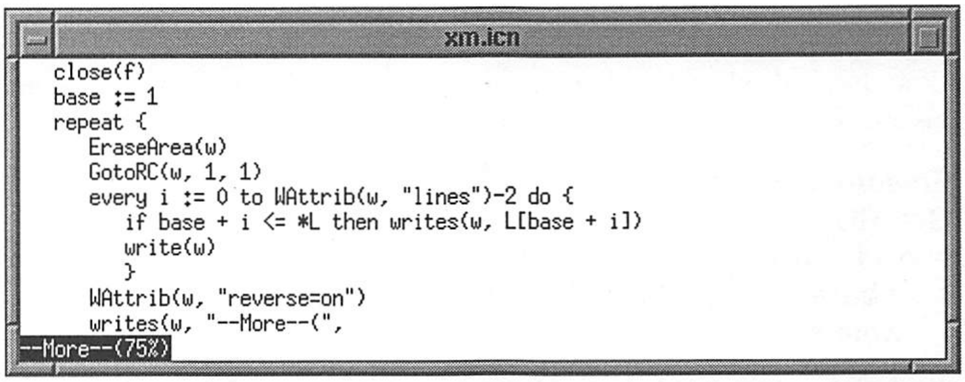
Fonts pose an even greater problem than colors for portable applications. Icon can use whatever fonts are available on a given system, but the application writer must determine what fonts to use on the systems the program will run on, or write code that works with user-selected fonts. To provide source code portability for typical applications, Icon defines four portable font names corresponding to the best available fixed- and proportional-width, serif and sans-serif system fonts. The portable names are typewriter, mono, serif, and sans and application programmers can expect them to be available in many type sizes.

Examples

Some simple examples illustrate various capabilities and the underlying philosopy adopted for graphics in Icon. It is beyond the scope of this paper to describe the Icon language itself, so the examples here rely on the reader's familiarity with C or Pascal. More exotic Icon-specific idioms are avoided; language constructs that are used and have no analogies in C or Pascal are explained. The major contentions supported by the examples are that a broad class of windowing applications need not be event-driven or based on callback procedures, and that interactive applications based on keystroke input need not be rewritten in order to incorporate mouse input. These examples are necessarily short; many more sophisticated examples are freely available in the Icon Program Library and can be obtained as described below in the section on Availability.

Xm: a File Browser

Xm is a trivial windowing version of the UNIX more(1) command. *Xm* displays a text file (its first argument) in a window, one screenful at a time, and allows the user to scroll forward and backward through the document. This simple version of *xm* is less flexible than more in most ways (it is written, after all, in less than thirty lines of code), but it gains certain flexibility for free from the graphics system: The window can be resized at any time, and *xm* takes advantage of the new window size after the next keystroke. Figure 1 is a sample screen image from *xm*.



```
close(f)
base := 1
repeat {
   EraseArea(w)
   GotoRC(w, 1, 1)
   every i := 0 to WAttrib(w, "lines")-2 do {
      if base + i <= *L then writes(w, L[base + i])
      write(w)
      }
   WAttrib(w, "reverse=on")
   writes(w, "--More--(",
--More--(75%)
```

Figure 1: An xm window

Xm begins with the lines

```
procedure main(argv)
   if *argv = 0 then stop("usage: xm file")
```

The main procedure starts by checking the argument count (unary * is Icon's size operator) and halting with an error message if the program has been invoked with no filename. A more robust version of *xm* would handle this case in the proper UNIX fashion by reading from its standard input.

After its (scant) argument checking, *xm* opens the file to be read, followed by a window to display it in. File mode "g" denotes a graphics window rather than a conventional file.

```
   f := open(argv[1], "r") | stop("can't open ", argv[1])
   w := open(argv[1], "g") | stop("no window")
```

If either of these values cannot be obtained, *xm* gives up and prints an error message.

With an open file and window in hand, *xm* reads in the file. It reads in all the lines at once and places them in a list. The loop terminates when a call to read() fails on end-of-file. A more intelligent approach would be to read the file gradually as the user requests pages. This approach makes no difference for short files but is superior for very large files. Variable base stores the index of the top line on the screen and is initialized to 1.

```
L := []
while put(L, read(f))
close(f)
base := 1
```

At this point, *xm* has done all of its preparatory work, and is ready to display the first page of the file on the screen. After displaying the page, it waits for the user to press a keystroke: a space bar indicates the next page should be displayed, the "b" key backs up and displays the preceding page, and the "q" key quits the application. The overall structure looks like:

```
repeat {
    # display the current page of text in the file
    # read and execute a one-keystroke command
    }
```

Xm writes out pages of text to the screen by indexing the list of lines L. The index of the first line that is displayed on the screen is remembered in variable base, and paging operations are performed as arithmetic on this variable.

```
EraseArea(w)
GotoRC(w, 1, 1)
every i := 0 to WAttrib(w, "rows") – 1 do {
    if base + i <= *L then writes(w, L[i + base])
    write(w)
    }
```

UNIX *more* writes a nice status line at the bottom of the screen in reverse video, indicating where the current page is within the file as a percentage of the total file size. Computing the percentage is done based on the last text line on the screen (base + WAttrib(w, "rows") – 1) rather than the first.

```
WAttrib(w, "reverse=on")
writes(w, "––More––(",
            ((100 > (base + WAttrib(w,"lines") – 2) * 100 / *L) | 100),
            "%)")
WAttrib(w, "reverse=off")
```

Keystrokes are read from the user using Icon's regular built-in function reads(). File functions such as reads() drop input activity that cannot be characterized in terms of characters. In order to enhance this application to support mouse input or explicit window resize handling, the call to reads() would be replaced by the more general function Event(), shown in the subsequent example programs.

```
case reads(w, 1) of {
    "q": break
    " ": base := (*L >= (base + WAttrib(w, "lines") – 1) | fail)
    "b": base := (0 < (base – WAttrib(w, "lines") + 1) | 0)
    }
```

Xm demonstrates that Icon demands little or no window system expertise of the programmer. Ordinary text applications can be ported to X with very few changes by adding a window argument to calls to functions such as read() and write(). After a program has been ported, it is simple to enhance it with features such as colors, fonts, and mouse handling.

Rfm: dragging a rectangle on-screen

Brad Myers challenged a CHI '92 workshop on language support for graphical user interface programming with the question: "How hard is it to make a rectangle follow the mouse around on the screen?" Obviously a toolkit builder can hardwire such an operation as a library function, but this begs the question and at the same time engenders large libraries with difficult conceptual hurdles.

Rfm is a trivial Icon program in which a rectangle follows a mouse. A window is opened via file mode "g", for graphics; a loop reads user input and erases the rectangle and redraws it on each drag event. There are no deep abstractions nor is control flow implicit or inverted; aside from lexical information the main graphical concept employed is that of reversible pixel operations, described below.

```
procedure main()
   &window := open("hello", "g", "drawop=reverse")
   while 1 do
      if Event() === (&ldrag | &mdrag | &rdrag) then {
         # erase box at old position, then draw box at new position
         FillRectangle(\x, \y, 10, 10)
         FillRectangle(x := &x, y := &y, 10, 10)
         }
end
```

The attribute "drawop=reverse" is Icon's portable form of XOR-mode reversible pixel drawing. In a classical XOR-mode drawing, the foreground color is XOR-ed with the contents of drawn pixels; the advantage is that repeating the operation erases what was drawn, while the disadvantage is that on color displays, XOR-ing the foreground color with current pixel contents results in an output pixel whose color is undefined. Icon offers "drawop=xor", but also provides "drawop=reverse", a drawing operation where the XOR of the foreground and background is XOR-ed with drawn pixels, guaranteeing that pixels whose current contents are the foreground color become the background color, and vice-versa.

There are some Icon operators that may not be obvious in this example, but are not unintuitive once defined. Ampersands preceding identifiers such as &ldrag are not address-of operators, but rather indicate the use of an Icon keyword. &ldrag, &mdrag, and &rdrag are constant values for mouse events, while &x and &y are the coordinates of the mouse at the time of the event returned by Event(). The use of keywords, rather than returning a structure, is analogous to the common practice in compilers of returning attributes in global variables during lexical analysis.

The backslash operator used in \x and \y is a null-value test which in this program causes the first call to FillRectangle() not to be performed the first time through the loop. The vertical bar | is an *alternation* operator; when read aloud it is pronounced similar to an

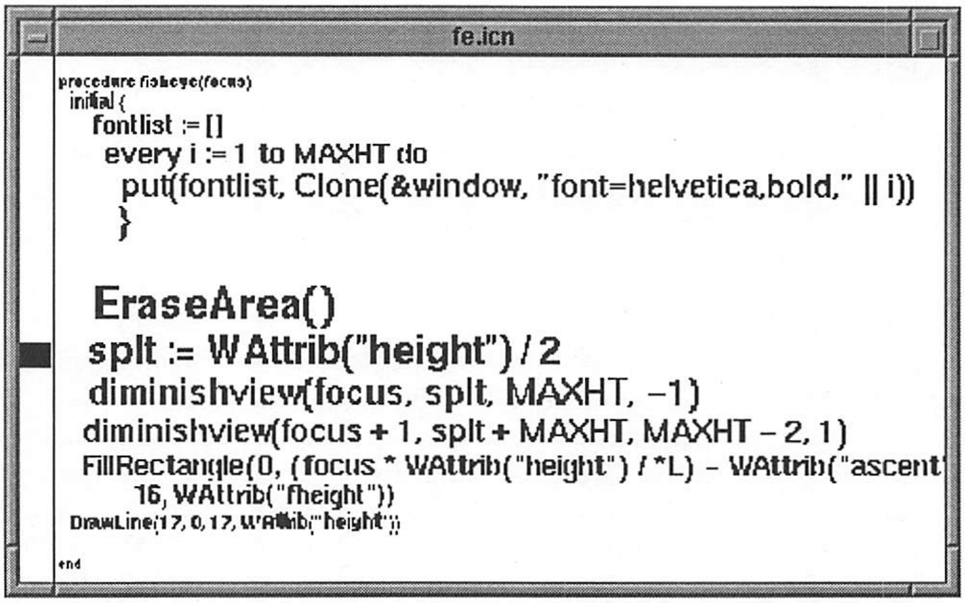
OR. Icon's goal-directed expression evaluation mechanism allows alternative values (the constants associated with left, middle, and right drags in this case) to be produced as needed in an attempt to satisfy the surrounding expression (in this case, to make the equality test true). In C the best equivalent code might look like:

```
e = Event();
if (e == LDRAG || e == MDRAG || e == RDRAG) { /* ... */
```

The conditional expression in Icon is somewhat more concise.

Fe: A Fisheye-View

A fisheye view is a visual metaphor in which distortion is used to emphasize important portions of a view by allocating more screen space to them [Furn86]. Screen space is allocated to a given element proportional to its distance from one or more focus points, computed in a domain specific fashion. Fisheye views are useful in a variety of contexts, such as maps. *Fe* is a simple fisheye viewer for text files, such as might be used to emphasize the current line of execution during debugging. A sample view is presented in Figure 2.



```
procedure fisheye(focus)
 initial {
   fontlist := []
    every i := 1 to MAXHT do
     put(fontlist, Clone(&window, "font=helvetica,bold," || i))
     }

   EraseArea()
   splt := WAttrib("height") / 2
   diminishview(focus, splt, MAXHT, -1)
  diminishview(focus + 1, splt + MAXHT, MAXHT - 2, 1)
  FillRectangle(0, (focus * WAttrib("height") / *L) - WAttrib("ascent
      16, WAttrib("fheight"))
 DrawLine(17, 0, 17, WAttrib("height"))
end
```

Figure 2: A fisheye view of some source code

The code for this example consists of a procedure main() and two subprocedures. The main procedure opens a window and reads an input file much as in the *xm* example presented earlier. The symbol MAXHT defines the pixel height of the largest text font, used to draw the line at the focus.

```
$define MAXHT 23
global fontlist, L
procedure main(args)
   fin := open(args[1]) | stop("no file")
   &window := open(args[1], "g") | stop("no window")
   flush(&output)
   L := []
   while put(L, read(fin))
   focus := *L / 2
   fisheye(focus)
   # ... main() continues ...
```

Fe's input handling consists of checks for mouse clicks in the scrollbar or "q" or the escape key to quit. Subprocedure fisheye() is called with the line number at which the focus (and the largest font size) should be directed.

```
   while 1 do {
      case Event() of {
         "q" | "\e": exit(0)
         &lpress | &ldrag: {
            if &x < 17 then {
               focus := *L * &y / WAttrib("height")
               fisheye(focus)
               }
            }
         }
      }
end
```

The fisheye procedure creates a list of drawing contexts for the window the first time it is called, each with a different font size. It then clears the screen and calls the helping procedure diminishview() to draw the text. Finally, a scrollbar is drawn on the left side of the window indicating the position and proportion of the viewing area within the entire file.

```
procedure fisheye(focus)
   initial {
      fontlist := []
      every i := 1 to MAXHT do
         put(fontlist, Clone(&window, "font=helvetica,bold," || i))
      }
   EraseArea()
   splt := WAttrib("height") / 2
   diminishview(focus, splt, MAXHT, -1)
   diminishview(focus + 1, splt + MAXHT, MAXHT - 2, 1)
   FillRectangle(0, (focus * WAttrib("height") / *L) - WAttrib("ascent"),
                 16, WAttrib("fheight"))
   DrawLine(17, 0, 17, WAttrib("height"))
end
```

The helping procedure diminishview() performs the actual text output, drawing each line in a successively smaller font size. It is called twice each time the screen is drawn, starting from the middle with the largest font and working towards the top or bottom of the screen as determined by the direction parameter.

```
procedure diminishview(focus, base, fontheight, direction)
    while 1 <= focus <= *L & 0 <= base <= WAttrib("height") do {
        GotoXY(20, base)
        writes(fontlist[fontheight], L[focus])
        base +:= fontheight * direction
        if 1 < focus < *L then {
            fontheight := max(fontheight – 2, 1)
            }
        focus +:= direction
        }
end
```

Experience

Experience writing graphics programs with Icon has revealed some basic aspects of graphics programming that are relevant to language designers. Flexible argument handling discussed in the preceding section—sensible defaults, automatic type conversions, and variable-length argument lists—has proven useful in graphics as it has in Icon's other application domains.

Another language feature that is of great utility in graphics is control structure heterogeneity. Icon control structures, most notably the case expression, allow clauses of any data type, and different clauses may be of different types in the same control structure. In the domain of graphics programming, this allows simpler code than other systems usually require.

The case expression is the natural multi-way selection construct in many programming languages, but in processing window system input, it poses a problem. Input consists of different kinds of data; key presses are fundamentally different from mouse actions. A key press is naturally represented as a one-character string with the ASCII value of the key pressed, but such a representation is not so appropriate for mouse activity. Various forms of control and function keys present a similar representation problem.

Rather than employing a variant record or some other indirect encoding, Icon represents key presses as strings and mouse actions as integers. Related information such as the mouse position at the time of the input is delivered via associated keywords, a technique similar to that commonly used for lexical attributes in compilers. Since Icon's case expression allows values of any type in the case-selectors, no variant record is needed and little or no mental effort is involved in decoding input events or processing mixed key presses and mouse events.

Conclusion

Icon uses a novel approach in the addition of graphics capabilities to a programming language with a conventional text-oriented input/output model. Icon succeeds in providing a smooth integration with pre-existing text operations and programming models. Experience with Icon has shown that simple windowing programs can be written in 10 to 100 lines, instead of the hundreds or thousands of lines required by many application program interfaces. Despite the limitations of a simplified programming model, Icon affords a concise notation with which to implement a wide range of graphic, window-based applications.

The approach used here to adding graphics to a programming language can be used for any high-level programming language. Some aspects of the design described here, such as the handling of colors and fonts, are directly applicable, while others fall into the category of making the extension fit naturally into the rest of language. In Icon, integration was facilitated by using capabilities such as generators and functions with an arbitrary number of arguments. Other languages offer different possibilities for the language designer.

Availability

A large library of programs and procedures written in Icon already exists and provides a resource of reusable code for future applications. Icon and its library are in the public domain. Icon is available by anonymous FTP to cs.arizona.edu; cd /icon and get READ.ME for navigation instructions. Icon Project Document 255 [Jeff94] is a complete description of the graphics facilities described in this paper and a compressed PostScript copy can be obtained by FTP in the directory /icon/doc, file ipd255.ps.Z.

Acknowledgements

Darren Merrill, Alex Jiang, and Barbara Wilkey contributed to the Presentation Manager, Macintosh, and MS Windows ports, respectively.

This work was supported in part by the National Science Foundation under Grants CCR-8713690 and CCR-8901573, a grant from the AT&T Research Foundation, and a UT San Antonio Faculty Research Award.

References

[Berk82] Berk, T., Brownstein, L., and Kaufman, A. A New Color-Naming System for Graphics Languages. *IEEE Computer Graphics & Applications*, pages 37–44, May 1982.

[Furn86] Furnas, G. Generalized Fisheye Views. In *CHI '86 Proceedings*, pages 16–23, June 1986.

[Jeff94] Jeffery, C. L., Townsend, G. M., and Griswold, R. E. Graphics Facilities for the Icon Programming Language; Version 9.0. Technical Report IPD 255, Department of Computer Science, University of Arizona, July 1994.

[Pike91] Pike, R. 8 $\frac{1}{2}$, the Plan 9 Window System. In *USENIX Summer '91 Conference*, pages 257–265, June 1991.

[Uhle88] Uhler, S. A. MGR — C Language Application Interface. Technical report, Bell Communications Research, July 1988.

Two Application Languages in Software Production

David A. Ladd
ladd@research.att.com

J. Christopher Ramming
jcr@research.att.com

AT&T Bell Laboratories

1 Introduction

PRL5 is an application-oriented language used to maintain the integrity of databases in the AT&T 5ESS™ telecommunications switch. PRL5 is unusual in that it was explicitly designed to eliminate a number of different coding and inspection steps rather than simply to improve individual productivity. Because PRL5 replaced an earlier high-level language named PRL, which in turn replaced a combination of English and C on the same project, it is possible to trace the effect of several fundamentally different languages on this single project. The linguistic evolution has been away from languages describing computation toward a "declarative" high-level language that has been deliberately restricted to accommodate the requirements of certain analyses.

Algorithms for checking database constraints are no longer specified by human developers; instead, code is generated from static representations of the constraints themselves. These constraint descriptions can be used in more than one way, whereas a program to check constraints is useful only for performing that particular computation. In effect, PRL5 allows the re-use of project information at a high level, before it has been specialized into particular implementations. The effects of this re-use on quality, interval, and cost are tangible. A key lesson is that application-oriented languages should not be designed to describe computation, they should be designed to express useful facts from which one or more computations can be derived.

2 Project Background

2.1 5ESS Database Constraints

The AT&T 5ESS is a high-capacity, exceptionally reliable digital switching system. The 5ESS software contains millions of lines of code produced and maintained by several thousand developers. At the heart of the 5ESS software is a distributed relational database that contains information about hardware connections, software configuration, and customers. For the switch to function properly this data must conform to certain integrity constraints. Some of these are logical constraints; for example, "call waiting and call forwarding/busy should never be active on the same line." Other constraints exist to document data design choices (redundancy, functional dependencies, distribution rules) that support efficient 5ESS operation and call processing.

2.2 Constraint Enforcement: Data Audits and Transaction Guards

5ESS integrity constraints are used in two kinds of software: data audits and transaction guards. *Data audits* check for all data violations; they are time-consuming because the database is large and there are many constraints. Data audits are useful for discovering accidental corruption as the switch is operating and for "cleaning up" switch data at strategically important points, such as when the 5ESS software is upgraded and includes data design changes. *Transaction guards*, on the other hand, ensure that incremental changes to the database leave it in a consistent state. In principle, transaction guards could be implemented by running a complete data audit before committing a transaction. In practice, complete data audits take far too long; transaction guards must be efficient — for instance, subscribing to features like caller-ID ought to be possible without having to re-verify every constraint. Fortunately, most transactions only add, delete, or change small

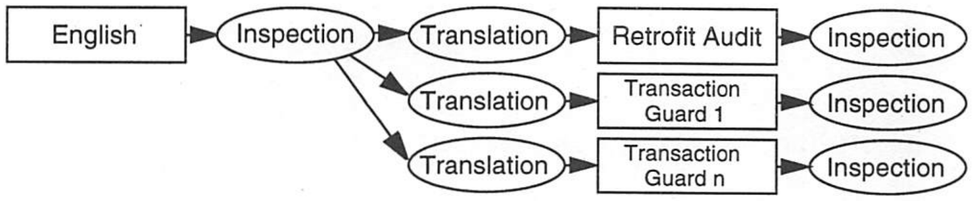


Figure 1: Traditional development with English specifications.

amounts of data and consequently have limited potential for violating constraints. If transaction guards are implemented properly and data is not modified without passing such guards, data should never become inconsistent and in theory data audits are unnecessary. However, to defend in depth against hardware and software failure, both audits are guards are used on the 5ESS.

Over the lifetime of the 5ESS, database constraints have been specified and implemented in three different ways. Initially, English was used to specify constraints; data audits and transaction guards were each written independently from these specifications. Data audits were used off-line to scrub data prior to installing a new software release; during switch operation, transaction guards were relied upon to maintain data integrity — there was no on-line data audit. In a second phase, an imperative high-level language called "PRL" was introduced. PRL is an acronym for Population Rule Language; putting data into the database is called "populating" the database, and the constraints themselves are called "population rules." PRL served simultaneously as the constraint specification language and the data audit implementation language. However, human coders continued to develop transaction guards independently using C. Now, a declarative database constraint language named PRL5 is used. Programs in the new language are executable specifications from which all constraint-related products can be generated: an off-line data audit, a new on-line data audit, a new "residual" data audit, and transaction guards.

3 Database Constraints in English and C

Originally, 5ESS development followed the standard software engineering practice of its day by specifying integrity constraints in natural language. But the gap between English and computer languages is large and results in difficulties that subsequent approaches strove to address.

One striking difficulty with English as a specification language is that it is ambiguous and lends itself to misinterpretation. Even seemingly straightforward requirements like those of figure 2 can be confusing.

For billing purposes, every telephone number must be associated with an active account.

No telephone number can have more speed calling destinations than have been paid for.

Figure 2: Some (contrived) English-language constraint specifications

The notion of a speed call button is familiar to many telephone users — it is a keypad digit used as an abbreviation for a longer "destination" number. Nonetheless, in the context of an implementation, the restriction on "destinations" in this constraint is unclear. Referring to the schema for relation *SpeedCalls* in table 1, it seems that this constraint could be implemented either as a limit on the number of *SpeedCalls* tuples associated with a particular number or as a limit on the number of distinct values in the "destination" fields in these tuples. Here, the choice is probably unimportant, but in a slightly different setting such ambiguities could result in costly errors.

A second pitfall of natural language specifications is that they cannot be compiled automatically and often diverge from their implementations. This problem was compounded in the 5ESS because each constraint was likely to affect several products: data audits as well as transaction guards. In the absence of an accurate mechanism to show the relationship between English specifications and C code, it is difficult to see which code needs to change when specifications are modified and vice-versa. As a result the code diverges from its specification, which in effect becomes "distributed" between the code and English text. Often the English text

RELATION NAME	ATTRIBUTES	KEY
SpeedCalls	number	Yes
	digit	Yes
	destination	No
Accounts	account	Yes
	status	No
TelNums	number	Yes
	owner	No
	maxspdcalls	No

Table 1: A sample relation schema

is abandoned to the role of documentation while the executable code is recognized as being more accurate. General-purpose code then becomes the sole repository of hard-won information — customer requirements, architectural decisions, and domain expertise — that is usually impractical or impossible to recover. In 5ESS, because the English constraints were implemented in several products, there was need for a "central point of truth." The English constraints therefore remained important, although over time the implementations were accorded increasing respect as specifications.

An additional problem, not entirely related to the use of English, was that no theory for developing transaction guards existed: only human intuition was available to achieve efficient results. To illustrate this problem, consider the constraint of figure 2:

For billing purposes, every telephone number must be associated with an active account.

Now suppose that a number is to be added or deleted. As mentioned earlier, it is possible to run a complete data audit (checking all telephone numbers for this property) before committing a transaction involving the addition or deletion of a number, but that would be too much unnecessary work. Assuming the database is correct before the transaction, a telephone number deletion should require no transaction guard at all. A telephone number addition should be guarded only by a check that the new number can be properly billed. But finding the smallest set of facts that should be checked to ensure continued data integrity is increasingly difficult as the transactions and constraints become more complex. In real life, correct results were rarely achieved (although this did not become clear until later) because specifications and code were not related by any known algorithm. Only a deeper understanding of integrity constraints and their relationship to transactions would lead to better results.

When the constraints were expressed only in English, quality problems could and did occur because of constraint ambiguity, "distributed" specifications, and the lack of an algorithm for generating transaction guards. Development took a long time, since the translation from English to various constraint implementations had to be done by human developers instead of a compiler (see figure 1). Costs were correspondingly high because human developers are expensive.

4 Imperative PRL

To eliminate the shortcomings of English as a specification language and thereby to improve interval, cost, and quality, a language called "PRL" [1] was developed. It addressed some but not all of the problems

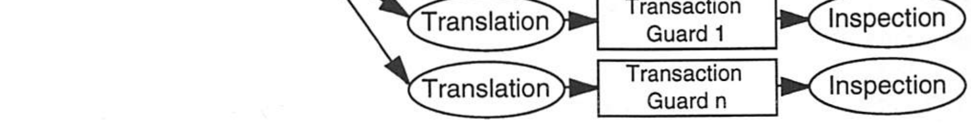


Figure 3: Software development with imperative PRL. Shaded portions are automated.

noted above. Experience with PRL is interesting because it reveals that general algorithmic languages, though powerful, have a limited ability to effect fundamental change in software development; by contrast, a restricted declarative language developed later proved more far-reaching in its consequences.

This original PRL was specially designed for implementing 5ESS data audits. It was an Algol-family language with variables, control flow, and functions and the "right" abstractions for the application domain. For instance, certain statements specified constraints that would raise exceptions when violated — such as "Accounts.status must_equal ACTIVE." It also contained control abstractions such as a "for every" statement that would iterate over tuples in a relation; the body of a "for every" statement typically contained assertions about the current tuple. This control abstraction eliminated the need for programmers to concern themselves with low-level database storage and access details. A "find" statement could perform relation searches for interesting tuples; a "must find exactly one" statement nested in a "for every" worked much like the join of relational algebra. PRL had a built-in knowledge of 5ESS data types — binary coded decimals, various kinds of enumerations and integers, strings, and the tuple types. The PRL compiler also made good use of knowledge about data distribution (over the 5ESS processors), and indices into 5ESS relations, reducing some burden on the programmers.

Figure 4 displays PRL implementations of the English-language constraints introduced in figure 2. Because they use appropriate abstractions and require little extraneous verbiage, imperative PRL programs

```
for every tuple in TelNums
begin
   must find exactly 1 tuple in Accounts
   where(TelNums.owner equals Accounts.account)
   when found
   begin
      Accounts.status must_equal ACTIVE;
  end
end

for every tuple in TelNums
begin
   int cnt;
   cnt = 0;
   for every tuple in SpeedCalls
   where(SpeedCalls.number equals TelNums.number)
   begin
      cnt = cnt + 1;
   end
   TelNums.maxspdcalls must_be_greater_than_or_equal_to cnt;
end
```

Figure 4: An imperative PRL specification

were easier for humans to read than programs in low-level languages such as C. This fact, coupled with the formality and precision of imperative PRL relative to English, caused one organization within the 5ESS to abandon English-language specification in favor of PRL programs that would serve simultaneously as code and specification. Of the two constraints implemented in the example PRL fragment, the first serves considerably better as a specification than the second, which is harder to analyze because it involves both state and control flow. PRL represents a compromise between English and C and was not an optimal specification language. The advantage of PRL, which must be measured against the cost of developing the language and tools, is that its programs can be compiled to produce data audits, eliminating a round of coding and inspection (see figure 3). For the data audit, this automatic code generation reduced interval and improved quality by eliminating errors of coordination and interpretation (although compiler bugs could still introduce faults). In addition, programmer productivity was boosted by the availability of primitive constructs at an appropriate level of abstraction.

In spite of its improvements, PRL suffered the plight of all general-purpose algorithmic languages: programs in such languages are opaque artifacts suitable only for interpretation on a particular machine. Interesting questions about programs in Turing-complete languages are often undecidable. In particular, the analysis needed to produce transaction guards is undecidable for Turing-complete constraint languages (like PRL) unless the transaction language is trivial. In essence, the constraints that were expressed in PRL could not be recovered for use in transaction guards. One implication is that the effect of general-purpose languages on large software development is limited because they can rarely be analyzed. Programs in such languages are a poor source of interesting information; in the case of PRL, this prevented the automatic generation of more than one product.

5 Declarative PRL5

Application-oriented computer languages need not be algorithmic; instead, they can describe facts relevant to a particular application. Given an appropriate algorithm, these facts can be used to generate the needed code. As an example, a yacc [3] program describes a formal language, but that description can be used to generate a parser as well as a diagram of the grammar and information about whether the grammar is ambiguous or whether certain productions are un-reducible. The grammar can also be easily transformed into a normal form. Even though yacc parsers can be augmented with reduction-time actions expressed in the host language, the grammar itself is often easily separated for use in other products. Just as the restricted declarative nature of the yacc language allows many different analyses, replacing imperative PRL with a restricted declarative language made it possible to generate data audits and transaction guards from a single source. This shift changed the development process, reduced costs, improved quality, and resulted in specifications that are a good source of information even for unexpected applications.

The missing piece of the puzzle was a procedure for deriving transactions from specifications. This was found in the work of Xiaolei Qian [4], who presented an algorithm (subsequently refined at AT&T[2]) for "differentiating" constraints to arrive at efficient transaction guards. The idea behind the algorithm is to find the weakest precondition of a given transaction with respect to the database constraints. By exploiting the assumption that the database is consistent (e.g., conforms to all constraints) before the transaction is executed, this weakest precondition can be factored into a form which in practice is less complex than the full weakest precondition. Then, at run-time, a transaction can be aborted if it fails to pass this simplified weakest precondition, thereby preventing inconsistencies from creeping into the database. The algorithm assumed a limited transaction language and also a constraint language based on first-order logic.

The new high-level language, dubbed "PRL5," was constructed on the strength of the "differentiation" algorithm and designed to conform to the requirements of that algorithm. PRL5 is based on first-order logic and does not describe computation directly — the notions of program counter, variables, and assignment are missing from the core language. But many of the abstractions useful in 5ESS constraint enforcement were retained by PRL5 language in some form — for instance, the "for every" and "find" control constructs were transformed into similar universal and existential quantification statements.

Figure 6 shows PRL5 implementations of the example constraints. Note that PRL5 programs describe constraints rather than an algorithm for checking whether constraints hold. Interestingly, the first constraint

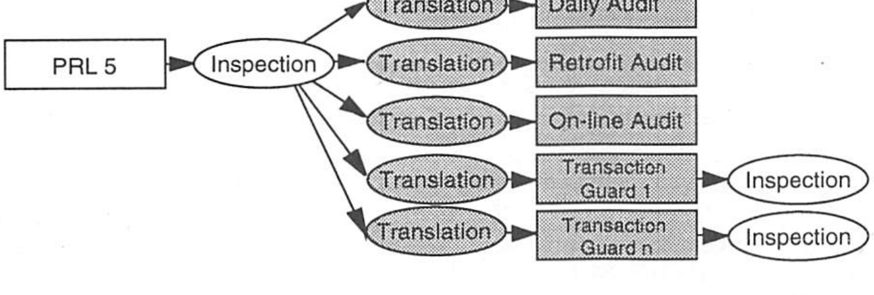


Figure 5: Development with declarative PRL5. Shaded portions are automated.

```
for number in TelNums
begin
  find >= 1 account in Accounts
  where(number.owner equals account.account)
  when_found
  begin
    account.status == ACTIVE;
  end
end

for number in TelNums
begin
  number.maxspdcalls >= count { s in SpeedCalls
                                where (s.number==number.number) };
end
```

Figure 6: A declarative PRL5 specification

looks similar to its imperative PRL equivalent. The important distinction is that the PRL5 "for" statement is defined in terms of universal quantification whereas the imperative PRL "for every" statement is really a looping control flow construct and can include "break," "continue," or other imperative statements such as variable assignment. The second constraint in figure 6, which formerly made use of iteration and stored information in a variable, has been rendered with a "count" expression. The subtle but important difference between this built-in aggregate operator and the invocation of a function call named "count" which simply hides imperative code is that PRL5 aggregates have been carefully considered for their ramifications on the "differentiation" algorithm (one of the important extensions of [2] accounts for such aggregates). Statements in PRL5 are limited to those which are "differentiable," whereas those in imperative PRL are not; PRL programs could check data constraints that cannot be expressed at all in PRL5. Situations that call for more complex constraints must be addressed by changing the assumptions of the system or reorganizing the data to arrive at a more tractable constraint.

```
find ==1 number in TelNums
where (number.number==new_tuple.num)
when_found
begin
   number.maxspdcalls>=1+count {s in SpeedCalls
                              where (new_tuple.num==s.num) };
end
```

Figure 7: Transaction guard for inserting a tuple into relation SpeedCalls

As an example of differentiation, consider the constraints of figure 6 and the simple transactions which either delete or add a tuple to relation SpeedCalls (there is a special transaction language offering inserts, deletes, updates, conditionals, and some iteration; realistic transactions and constraints are usually more complex than those in this example). Deleting from the relation SpeedCalls cannot possibly violate any of the constraints in figure 6. Adding a SpeedCalls tuple involves the check of figure 7.

Note that the variable `new_tuple` is not bound in this check and is a value that must be provided at run-time. An inspection of this transaction guard reveals that an expensive recount is required each time a SpeedCalls tuple is added. A better data design would store the number of tuples in a new attribute and one would expect that differentiation would then yield better results. Such a counter could be added to the TelNums relation as shown in Table 2. A constraint relating numspdcalls to the tuples in relation SpeedCalls

RELATION NAME	ATTRIBUTES	KEY
TelNums	number	Yes
	owner	No
	maxspdcalls	No
	numspdcalls	No

Table 2: A modified "TelNums" relation

owned by the number would need to be added; a check limiting the number of SpeedCalls can then refer to a stored value rather than a computed one. The revised check is shown in figure 8. An auxiliary constraint requires each tuple in SpeedCalls to be related to an existing TelNums tuple.

```
for number in TelNums
begin
   number.numspdcalls<=number.maxspdcalls;
   number.numspdcalls==count { s in SpeedCalls
                              where (s.num==number.number) };
end
for speedcall in SpeedCalls
begin
   find ==1 number in TelNums where (number.num == speedcall.num);
end
```

Figure 8: Constraints revised to reflect the new data design

With these revised constraints, the differentiator will no longer allow a transaction to delete or add to relation SpeedCalls alone: it is necessary to update the TelNums relation simultaneously. Because the differentiator "understands" aggregates such as count, neither adding nor deleting tuples from SpeedCalls will require iteration over the SpeedCalls relation; the transaction guard is merely that of figure 9. This analysis lies at the heart of the PRL5 language, which was designed precisely to accommodate the algorithm that performs it.

```
find ==1 number in TelNums
where (number.number == new_tuple.number)
when_found
begin
   number.numspdcalls+1<=number.maxspdcalls;
end
```

Figure 9: A more efficient transaction guard

5.1 The Applications of PRL5

Like yacc code, which has several potential applications, PRL5 code can be (and is) employed in several different tasks. Most importantly, PRL5 is compiled to produce code for several products. Two kinds of data audit — one for use on-switch and one for use off-line — are produced from PRL5 specifications. Numerous transaction guards have been derived, which is particularly significant since they are for the first time being developed in a provably sound fashion. A hybrid partial audit called a "residual check" or "daily audit" is also derived from PRL5 code to double-check the embedded base of transactions developed with the old methodology at the end of each day; this "residual check" is new and has been made possible by the versatility of declarative PRL5 specifications.

In addition, because PRL5 is declarative, it is also the source of some lesser but nonetheless useful analyses that would not have been possible if PRL5 had been a general-purpose language. In particular, PRL5 offers a view of the data design that augments a basic database description enumerating relations, fields, keys, and indices. The relationships between data items can be understood by examining PRL5 constraints for such information as foreign key constraints and functional dependencies; software visualizations revealing this information guide future data design and feature implementation.

PRL5 can also be "optimized" using techniques often unavailable or of limited value when applied to imperative code. Redundant code elimination, logical and semantic transformations, and simplifications are all good optimization candidates. For instance, the absence of state makes "loop jamming" effective for combining separate constraints quantified over the same relation; this results in an implementation that traverses a relation just once instead of many times. Another useful optimization involves the automatic selection of a good lookup method given index information, knowledge about keys, and the ways in which keys are populated. One compelling example involves lookups where keys are partially specified. Depending on certain factors, it may be more efficient to probe for each possible value with a keyed lookup rather than perform a linear search. Which method to use is never specified within PRL5 code; rather, the choice is made by a compiler. If the factors affecting the decision change, an improvement can be effected without relatively risky modifications to the PRL5 constraints themselves.

Another interesting aspect of PRL5 "code" is that terse error descriptions can be extracted at compile-time and presented when constraints are violated. In a general-purpose algorithmic language, a constraint violation can be identified crudely at best, for instance by giving the line number of the failed assertion. But in PRL5, it is possible to exhibit precisely the code that is necessary and sufficient to describe the violated constraint, making the error easier to understand and to fix. These error constraints, being valid PRL5 programs in their own right, were formerly "verbosified" by a program which rendered the constraint in English for the benefit of those unfamiliar with PRL5 syntax; however, lately PRL5 code itself has proven adequate for most descriptive purposes without this additional transformation.

5.2 PRL5 Impact

The impact of PRL5 on software development has been to eliminate certain major coding and inspection steps while at the same time offering an increased number of products. These improvements are reflected in figure 5. More constraint-related products are now offered, as compared with the situations detailed in figures 3 and 1; other things being equal, this is a form of productivity improvement. The elimination of coding and inspections affects cost and interval. The single source of constraint information and the technology for automatically compiling these constraints both result in quality improvements.

Although PRL5 was intended to eliminate several development steps, transaction guards continue to be inspected by people even though they are automatically generated (accordingly, this step is drawn with white nodes in figure 5). These inspections prove necessary because differentiation results sometimes surprise developers by being "too inefficient" or, occasionally, outrightly impossible to satisfy. These bad results indicate previously unnoticed problems with the data design, the transaction, or the constraints themselves. Differentiation therefore acts as a useful "sanity check" as well as a code generation step; its results must be heeded. For instance, the transaction guard of figure 7 prompted a small change to the data design that resulted in the better guard of figure 9. Furthermore, if (given the new design) a transaction attempted to add or delete a SpeedCalls tuple without updating "numspeedcalls" in the appropriate TelNums tuple, this mistake would have been revealed by inspecting the transaction guard output. The curious result is that an intended improvement in development interval and cost resulted instead in a quality improvement.

The declarative form has had a more profound effect on development than any imperative language could, because PRL5 is a more useful repository of information. More products can be generated automatically, and time-consuming coding steps as well as inspections have been eliminated. Quality is higher, partially because problems of coordination have been eliminated and partially because the differentiation algorithm provides useful new information.

6 Conclusion

The 5ESS database constraint components have evolved significantly since the days when English was considered a suitable specification language. The second and third generations of this projects show two "very high level" languages in stark contrast. The first, being Turing-complete, was effective for describing algorithms to enforce constraints but not the constraints themselves; its use therefore was limited to straightforward execution on a particular machine. The new language, PRL5, is geared toward the description of constraints, and its programs can be used to derive computation for enforcing constraints automatically in a variety of ways. This suggests that although computation is the natural purpose of any computer language, languages that describe computation directly are not always the most useful ones because they tend to be impossible to analyze — they hide information rather than expose it.

The application language approach is to design a domain-specific language that can be analyzed to yield executable code as well as secondary applications. The advantage is in explicit representation of domain information that would otherwise be implicit and inaccessible; this approach reduces the need for alternative documentation in languages such as English, and leaves open the possibility that unforeseen uses of the information can be found. The drawback, of course, is that designing, implementing, and introducing new languages is difficult. Although PRL5 eliminated a number of tasks previously executed by humans, some of these improvements were offset by shifts in resource allocation due to the integration of this new language. Nonetheless, as of this writing, PRL5 successes include the deployment of new on-switch data audits, the automatic generation of transaction guards, and quality improvements that are having a positive impact on the 5ESS.

Acknowledgements

The work described in this paper was the result of a collaboration between several organizations. Many individuals made significant contributions to the project. Rather than running the risk of omission, we will refrain from listing them all here. A smaller and more manageable number of people offered valuable insights and advice leading to this paper. For this we thank gratefully thank Jackie Antis, Michael Benedikt, Jon Bentley, Tim Griffin, John Meale, John Pyrce, Eric Sumner Jr., and Dave Weiss.

References

[1] B. N. Desai, D. L. Harris, and R. A. McKee. A formal language for writing data base integrity constraints. In *International Switching Symposium*, 1992.

[2] T. G. Griffin and H. Trickey. Integrity maintenance in a telecommunications switch. *IEEE Data Engineering Bulletin*, June 1994.

[3] S. C. Johnson. Yacc: Yet another compiler compiler. Technical report, Bell Telephone Laboratories, 1975.

[4] X. Qian. *The Deductive Synthesis of Database Transactions.* PhD thesis, Stanford University, 1989.

Chris Ramming received degrees in Computer Science from Yale College (BA '85) and the University of North Carolina at Chapel Hill (MS '89). He joined AT&T Bell Laboratories in 1987 and is a Member of Technical Staff in the Software Production Research department. His current interests include application languages and their use in software production.

David Ladd received the BS and MS degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1987 and 1989. He joined AT&T in 1989, where he is currently a Member of Technical Staff in the Software Production Research Department. His current research interests are software re-engineering and application-oriented languages and environments.

Using a Very High Level Language to Build Families of High Quality Reusable Components

Gary F. Pollice
CenterLine Software, Inc. and
U. Massachusetts, Lowell

1 Introduction

While most programs are designed to perform a specific task, they have a natural evolution over time; causing a single program to become a set of programs that perform related tasks. In 1976 David Parnas introduced the concept of a *family of programs* [11]. He suggests that any program should be considered a member of a family of programs, all of which perform related tasks. If one plans for change when a program is designed less effort may be required to produce future revisions.

Today there is an emphasis on building software components for reuse. Components may be designs, programs, functions, classes, or code segments. Major reuse projects are in progress at several institutions, for example, the Software Productivity Consortium and the Software Engineering Institute [1, 6]. Processes have been developed to introduce and support reuse programs. The disciplines of domain engineering and software synthesis emphasize creating families of components and developing tools to assist in constructing them. Component generation is a prominent part of most efforts.

Until recently, the literature has focused on the processes. Generators are assumed to be available; however, good descriptions of what a generator looks like and how to build one have been scarce. Some exceptions do provide insight to useful generation techniques [13, 4, 5, 2].

The work described in this paper is a continuation of work done by the author [12] in which a framework for building component generators is presented. The goals were to be able to produce members of a family of components, each as good as hand coded components, and to determine if the techniques and architecture would generalize and scale up. The example used is a family of scanners, collectively known as **scangen**, for C and C++.

Metaprogramming, that is writing programs to write programs, is fundamental to component generation. Very high level languages (VHLLs) have been shown to be useful for metaprogramming [13, 4]. **perl**[14] is the VHLL used in the work described.

The remainder of the paper contains:

- a description of the problem for which generators were designed and a discussion of alternate approaches.
- a presentation of the generator architecture and design,
- a discussion of the resulting components, **perl** as a generator building language, and the benefits obtained, and
- a description of on-going and future work

2 The Problem

Language processing components like scanners are used in several settings. The components can be found in traditional compilation systems, editors, browsers, file filters, and other applications found in program development environments. Varying requirements are placed upon the components.

The scanner requirements for **scangen** include handling different character sets, varying the storage model (i.e., having the scanner be completely reentrant), changing the target and input language dialects, execution on different computing platforms, and variable amounts of information to be included in the token abstractions.[1] The ability to scan languages other than C or C++ is not a requirement. An additional desirable property, but not a requirement, is to have the scanner's source code (the target language) be either C or C++. Traditional scanner generators like **lex** and **flex**[8] are unsuitable solutions to the problem described. They allow one to change the input language, but little else. **flex** allows more variation than **lex** but it still does not address the requirements well.

Several options may be considered for implementing the required scanners. They are listed here with comments on their suitability for the task.

- modify an existing scanner generator — This is a labor intensive solution. It was rejected because of the amount of programming required and for the anticipated amount of debugging.
- write individual scanners — While it is not very hard to write a C or C++ scanner, doing it several times, with minor modifications is tedious. Even if code were reused a lot, propagating changes and bug fixes to many versions of a program invites problems.
- write one scanner to handle all variabilities — This approach was rejected because the resulting scanner would be too complex, filled with conditionally compiled code that would make it hard to read, and would likely not have the performance demanded for some of the required environments.
- develop a class library for scanners — This approach seemed attractive at first. It was rejected because the approach only allows customization by expansion which makes it difficult to avoid *software bloat* and maintain a small memory footprint where required. This approach also requires scanner modification for use in a C application.
- build a scanner generator that allows for wide variability — This is the approach taken and described in the rest of the paper.

3 Generator Details

3.1 Types of Variability

A component generator must be able to deal with three types of variability: specified constants, simple computations, and parameterized computations. The language used to program the generator should allow one to easily express and implement each type.

A specified constant is a value defined in the specification and given to the generator. A specified constant occurs when a name, such as an identifier or type, is constant in a component but may vary across members in a component family. One uses specified constants for textual substitution during generation. Examples of specified constants in the scanner generator are the file names and extensions of the generated modules.

A simple computation is a computation that occurs once during a single generation. The value computed is constant throughout the generation. A simple computation differs from a specified constant in the way its value is determined. The generator computes the value based upon the specification. Examples of simple computations in the scanner generator are the name of the character type and the values in the `lexemeID` table. The value of the character type depends upon the input character set specified. Values in the lexeme table depend upon the input language, input language dialect, and other characteristics described in the specification.

A parameterized computation is a computation that occurs more than once during a single generation. The value of the computation depends upon values supplied to it by the generator. One uses a parameterized computation to produce program parts that have a standard form with changing values. An example of a parameterized computation is a one that produces a function

[1]For example, some scanners were required to pass complete source information through to the parser so the source program could be recreated. Others were only required to pass the token code and text of identifiers and strings.

declaration. The name of the function, its type, and parameter types and names will vary each time the computation is performed.

3.2 The Scanner Generator Organization

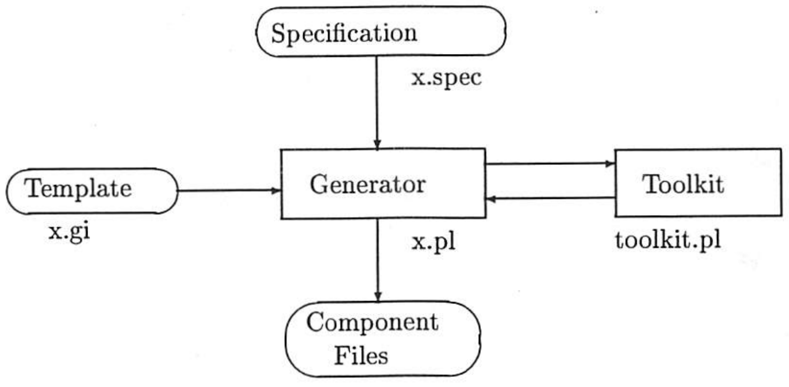


Figure 1: Generator organization.

Several C and C++ scanners were analyzed to identify the common and variable parts. After the analysis, the generator organization shown in Figure 1 was developed. Table 1 briefly describes each of the five components. A more detailed description of each is presented in later sections.

Part	**Contents**
Specification	Specification of the generated component. The specification consists of definitions and values required for generating the variable parts of the component.
Toolkit	General purpose generator functions and procedures. This file is typically limited to specific target languages.
Template	Common source code segments with embedded computations for variable parts.
Generator	Component family specific functions and procedures — the control module for the generator.
Component files	Generator output — C or C++ source code.

Table 1: Contents of a generator's components.

Not all of the generator parts in Figure 1 are required. In some cases there may be no template file or there may be multiple template files. The token module of the scanner generator has no associated template file. The token abstraction is a structure or class containing fields for the token code, string text pointer where applicable, and file position information. Fields may be added or removed, depending upon the application. The generator toolkit automatically generates constructors, destructors, and access functions for structured types if the option is selected in the specification.

3.3 Selecting a Language for Writing the Generator

Metaprogramming is mainly an exercise in text processing where the text is the source code of the resulting programs. In order to implement a source code generator, a VHLL with excellent string manipulation functionality is required. In addition, the VHLL must possess enough computational power to deal with the types of variability described in Section 3.1. In order to support template processing, the language should have the ability to evaluate programs during generation.

Three VHLLs were considered for the scanner generator work: **awk**[7], **perl** (version 4), and **Tcl**[10]. Each is appropriate for programming a generator. **perl** was chosen for several reasons. In addition to its strong string manipulation capability it has parameterized subroutines, an `eval` function, and a rich set of libraries. Most important to the work, **perl** is available on the platforms used (Macintosh and UNIX workstations[2]) and has a debugger.

3.4 The Scanner Specification

The scanner specification is a **perl** source file. The specification consists of a set of assignment statements. The assignments initialize variables with values that completely define the resulting scanner Listing 1 shows a part of a specification for a scanner that is written in ANSI C and accepts ANSI C, including multi-byte characters as described in the ANSI standard for the C language [3].

```
# Information about the language the scanner will accept.
$inputLanguage = "C";                       # C or C++
$inputDialect = "ANSI";                     # ANSI or nonANSI
$charSet = "multibyte";                     # ASCII, multibyte, etc.
$OK8bits = $FALSE;                          # TRUE => 8-bit chars. in identifier
$oldStyleOps = $TRUE;                       # TRUE allows =>, =+, etc.
$allowDollar = $FALSE;                      # TRUE => Allow '$' in identifiers
$maxIDLength = 50;                          # Max. # of chars to keep for
                                            #   identifiers.
###############################################################################
# Information about the target language .
$genTargetLanguage = "C";                   # C or C++
$genTargetDialect = "ANSI";                 # ANSI or nonANSI
###############################################################################
# Other variables that control what things get generated and other values.
$generateStrings = $TRUE;                   # TRUE => gen. string abstraction
$generateTokens = $TRUE;                    # TRUE => gen. token abstraction
$generateTests = $TRUE;                     # TRUE => generate test info.
$generateAccessFunctions = TRUE;            # TRUE => generate access functions
                                            #   for data members.
$stringTableSize = 16384;                   # Initial string table size
$stringTableIncrement = 4096;               # How much to grow the string table
                                            #   by when necessary.
$hashSize = 512;                            # Number of hash slots
                                            #   must be power of 2.
$scanErrorFunction = "";                    # User supplied error function.  If
                                            #   an empty string, then ScanError
                                            #   outputs the message.
$tabExpansion = $TRUE;                      # $TRUE => expand tabs for counting
                                            #   column numbers
$tabStop = 8;                               # Number of spaces per tab stop
```

Listing 1. Part of a scanner specification.

3.5 The Generator Toolkit and Templates

The generator toolkit supplies general purpose functions to domain specific generators. There are over forty functions in the toolkit. The functions can be broken down into groups that perform the following tasks:

- formatting source programs,

[2]UNIX is a registered trademark of UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc.

- creating structured and enumerated types, and declarations,
- formatting function declarations, function calls, parameters, and arguments, and
- evaluating templates.

There is one generator toolkit used for the scanner generators. Differences between C and C++ target languages are processed by the toolkit routines. A set of toolkit variables are exposed to the other generator modules. The modules can set these variables to control the toolkit behavior. The variables, *$genTargetLanguage* and *$genTargetDialect* in Listing 1 are two such variables.

Listing 2 shows a typical generator toolkit function, *&New*, which generates code to create a new instance of an object. If the target language is C++, the constructor for the class is used. If C is the chosen target language, a function is called which returns a pointer to the appropriate object. The C function name is made up from the name of the type prefixed with the word "New". Consistent style is imposed and maintained by the toolkit code.

```
sub New
{
    local($type, $params) = @_;

    if ($genTargetLanguage eq "C++") {
        if (@_ == 1) { return "new $type"; }
        else { return "new $type($params)"; }
    } else { return "New$type($params)"; }
}
```

Listing 2. The *&New* function from the generator toolkit.

An effective template mechanism is one that allows values to be expressed in the template that can be computed at generation time. This requires the generator (and the language in which the language is written) to be able to read embedded statements, evaluate them, and substitute their output for them in the template. The *&GenEvalFile* function shown in Listing 3 is the toolkit routine that performs this operation. It works in conjunction with a *&ReadLine* and *&WriteLine* function to read in a template and output the generated code to the proper file.

Three things are done by *&GenEvalFile* on template text lines:

1. If there are no embedded **perl** statements or embedded comments, the text is written to the output with no modification.

2. If there is an embedded comment, convert the comment text to the proper format for the target language and output the comment. Embedded comments in the template begin with '`@@`' and continue until the end of the line.

3. If there is embedded **perl** code, evaluate it and replace the input text with the results of the evaluation for output. Embedded **perl** code is enclosed within '[[' and ']]' delimiters and may be continued on extra lines.

Sample template code is shown in Listing 4. The template is for the *DepositChar* function in the string table module. The example illustrates variable substitution and replacing a function call with the text it produces (i.e., *&Params* to format parameters properly). The last line of the listing causes an empty string to be emitted except when multibyte characters are accepted; in which case the *DepositWChar* function is emitted.

When a scanner is generated using the specification from Listing 1, the code shown in Listing 5 results when the previous template is evaluated.

```
sub GenEvalFile
{
    local($i, $t, $cmd);

    while (&ReadLine) {
        $i = index($inbuf, "[[");
        while ($i >= $[) {                       # Start of an embedded command.
            $outbuf = substr($inbuf, 0, $i);
            &WriteLine;                          # write out to start of command
            $t = substr($inbuf, $i+2);           # build up the command
            $i = index($t, "]]");
            while ($i < $[) {
                $t .= &ReadLine;                 # end not here, keep reading
                $i = index($t, "]]");
            }
            $cmd = substr($t, 0, $i);
            $outbuf = eval($cmd);                # replace the input text
            &WriteLine;
            $inbuf = substr($t, $i+2);
            if (length($inbuf) == 1) { $inbuf = ""; }
            $i = index($inbuf, "[[");
        }
        $i = index($inbuf, "@@");
        if ($i >= $[) {
            $outbuf = substr($inbuf, 0, $i);
            &WriteLine;                          # up to the comment
            $inbuf = substr($inbuf, $i+2);       # remove @@
            chop($inbuf);                        # remove ending newline
            $inbuf = &Comment($inbuf, 0) . $NL;
        }
        $outbuf = $inbuf;
        &WriteLine;
    }
}
```

Listing 3. The *&GenEvalFile* function for evaluating template files.

```
[[$genVoidType]] DepositChar[[&Params("$charType", "c")]]
{
    if (stix >= (stLimit - 4)) {                @@Getting close to limit
        stringTable = (char *)realloc(stringTable,
            (stLimit + [[$stringTableIncrement]]) * sizeof(char));
        stLimit += [[$stringTableIncrement]];
        if (stringTable == NULL) {              @@Error in realloc
            scanAbort = 1;
            ScanError("Error reallocing string table");
            return;
        }
    }
    if (stringTable == NULL)
        return;
    stringTable[stix++] = (char)c;
}
[[$DepositWCharBody]]
```

Listing 4. Template for the *DepositChar* string table function.

```
void DepositChar(wchar_t c)
{
    if (stix >= (stLimit -4)) {                /* Getting close to limit */
        stringTable = (char *)realloc(stringTable,
            (stLimit + 4096) * sizeof(char));
        stLimit += 4096;
        if (stringTable == NULL) {             /* Error in realloc */
            scanAbort = 1;
            ScanError("Error reallocing string table");
            return;
        }
    }
    if (stringTable == NULL)
        return;
    stringTable[stix++] = (char)c;
}

void DepositWChar(wchar_t c)
{
    char        *cp;
    int         mbCount;

    if (stix >= (stLimit - 4)) {
        stringTable = (char *)realloc(stringTable,
            (stLimit + 4096) * sizeof(char));
        stLimit += 4096;
        if (stringTable == NULL) {
            scanAbort = 1;
            ScanError("Error reallocing string table");
            return;
        }
    }
    if (stringTable == NULL)
        return;
    cp = &(stringTable[stix]);
    mbCount = wctomb(cp, c);
    stix += mbcount;
}
```

Listing 5. Code generated for *DepositChar*.

3.6 Escapes

There are cases where a generator is either insufficient for a specific application but will be close enough to be desirable. One wants to avoid modifying generated code. Once the code has been modified, it must be re-modified each time the component is generated. Escapes are included in a generator to handle such cases by allowing user supplied code to replace generated code.

scangen allows specific functions, like the error reporting function, to be supplied from elsewhere. A default error function is generated which outputs a message on `stderr`. However, if the error function is escaped, the calls to it will remain in the generated code, but the routine must be supplied by the programmer. The name of the error function is stored in the *$scanErrorFunction* variable shown in Listing 1.

Complete modules may be omitted from generation, just like functions. In the **scangen** case, each of the string table, I/O, and token modules can be supplied rather than generated.

Languages like **perl** support escapes well. Since they are interpretive functions are only required when they are called. Therefore, if a generator never calls a function it does not have to be supplied. If the feature were not present in the VHLL, default *stub* functions would have to be provided and would have to be overridden when the functionality is required.

3.7 Name Generation

A particularly effective use for generators is to generate names and values for objects in a consistent, understandable manner. In a scanner, token codes must be assigned to each possible entry in the lexicon. Some of these are generic names, like "identifier" while others are specific to one token, like the "!=" operator in C. Generating the names and assigning values to them insures consistency throughout the scanner. This is a particularly error prone operation when manually programming multiple scanners.

```
@archaicOperators = (
"=*", "=/", "=%", "=+", "=-", "=&", "=^", "=|",
"=>", "=<", "=>>", "=<<"
);

##############################################################################
# The following pairs are used to give names to the symbols used
# in making up operators, etc.
#
%tokSymbolNames = (
"_", "under",
"=", "eq",
'"', "quote",
"*", "star",
"%", "pct",
.
.
.
```

Listing 6. Arrays used for naming token codes.

Languages like **perl** that support associative arrays are effective for name generation. The strategy used in **scangen** is to have the base name for each of the token codes stored in one of several arrays. The arrays contain a subset of the possible tokens and each is used as required by the input specification. An additional associative array is used to assign a word to each possible operator character. When a token code name is required, if the base name is a word, the token code prefix, `TC` in this case, is prepended to it. If the base name is an operator, a function is called to turn the operator into a, possibly compound, word and then the token code prefix is prepended.

Listing 6 illustrates the array of archaic C operators and the beginning of the symbol name array. If the token code for the symbol `=*` is required, the *&MakeTokenWord* function, shown in Listing 7 is called. It returns the name `TCeqstar`.

```
sub MakeTokenWord
{
    local($symbol) = @_;
    local($i) = 0;
    local($r) = "TC"; # the result

    while ($i < length($symbol)) {
     $r .= $tokSymbolNames{substr($symbol, $i, 1)};
$i++;
    }
    $r;
}
```

Listing 7. *&MakeTokenWord*, a function for creating token names.

4 Results

Results of the scanner generator work is described in the following sections. A couple of versions of the generator have been produced which have generated dozens of scanners. Several areas of consideration are discussed.

4.1 Productivity Improvement

One way to determine the usefulness of a generator is to measure the ratio of generator code to generated code for one component. For instance, generating programs to input a description of a deterministic finite automaton (DFA) and produce a C++ program that represents the DFA has yielded a ratio of generator code to generated C++ program code of about $1:4$ [9]. The scanner generator, not including the code for the generator toolkit, yields a ratio of about $4:3$.

If lines of code is the only metric used, one would judge the scanner generator a failure. However, many scanners can be generated by the generator. If, for instance, eight different scanners are generated by the scanner generator we have a ratio of $1:6$ (i.e., $4:8*3$).

If one were to hand code multiple versions of a scanner, pieces of code would be used in a cut-and-paste fashion. However, multiple versions of the scanner would have to be maintained. The duplicated maintenance work is eliminated with the generation approach.

One of the biggest productivity gains came in regression testing. The output of a component generator is a source program. Modifying the generator to produce new members of the component family should not change previously generated source programs in any substantial way. Changes to source programs are easy to locate and the detection takes little time compared with re-running a test suite.[3] Since there is no conditionally compiled code in the generated source programs, features that are not appropriate to a particular scanner are never part of its source code.

4.2 Performance

The goal was to produce scanners that were as efficient as hand coded ones. Since the code generated is almost identical to the hand coded scanners used as models, the goal was achieved.

A scanner generated by **scangen** has been compared to a scanner generated by the **lex** and **flex** scanner generators. The results are shown in Table 2. All scanners were generated on a Sun SPARCstation ELC under SunOS 4.1.3. The **lex** version is the one shipped with the operating system. **flex** version 2.3 was used for the comparison. Three versions of a **flex** scanner were generated. These represented the extremes in the space-time tradeoff spectrum and a middle of the road version. The **scangen** scanner used non-generated token and I/O modules. This was done in order to make sure similar scanners were being compared. Each scanner used the standard input stream and returned a simple integer token code for each token recognized. The times shown are the average over several runs of scanning an approximately 50,000 line C++ source file. Times are shown in seconds. The program size is the sum of `text`, `data`, and `bss` as shown by the **size** program.

4.3 Maintainability

Multiple versions of the scanner generator have been produced. This is not a desirable feature. One would like to maintain just one generator. However, the generator's complexity grows as quickly as any comparably sized program. Therefore, while the maintenance effort is not any greater (and is, in fact spread out over the number of components generated), maintenance is not, however, any easier with the type of generator described.

Generators need to be expanded and contracted, just like any other family of programs. The generators described are single points that generate a family of components. A more desirable

[3] It takes about five seconds to generate a scanner with the scanner generator and five additional seconds to compare the generated source modules with previously generated versions. It takes several minutes to run a typical test suite for C scanners.

Scanner	Real time	User time	Program size
scangen	4.8	4.5	22888
lex	10.6	10.3	30368
flex	5.4	5.2	16384
flex -C	5.0	4.7	24576
flex -Cf	3.3	3.1	106496

Table 2: **scangen** scanner performance vs. standard generators.

architecture is one that has a family of generators, each of which generates a family of related components. This is further discussed in Section 5.

Use of the **scangen** generator beyond the initial implementation led to a major modification. The original generator that generated four modules was broken up into four generators, each generating one module. Each generator uses the same input specification.

4.4 perl as a Generator Implementation Language

For most things, **perl** was a good choice for the generator implementation language. It allowed changes to be implemented and tested quickly and the debugger was invaluable.

perl is a big language which makes mastery difficult. The documentation to date is barely adequate. A lot of time was spent trying to figure out how to get something to work properly. In most cases there was more than one way of getting the job done. The advantages and disadvantages of each were not clear.

Since metaprogramming tasks are mostly string manipulation, **perl** was more than adequate for the work. One would like to write code segments in the generator as close as possible to the way they appear in the generated code. Variable substitution in strings helps make this possible. However, in order to handle strings generated from parameterized computations, a desirable feature of an implementation language would be to allow function calls to be replaced with the result of the function inside of strings (or provide a special string type that would allow this).

For example, in the token module, the following line of code appears:

```
$scanReserve .= "     Reserve( " . &str($cppKeywords[$i]) . ", TC$cppKeywords[$i]);\n";
```

A more desirable line of code is:

```
$scanReserve .= "     Reserve( &str($cppKeywords[$i]), TC$cppKeywords[$i]);\n";
```

5 Future Work

The problem of generator maintainability, extensibility, and contraction was mentioned in Section 4.3. Current work is focusing on being able to formally describe the generated family of components. Such descriptions require common and variable parts to be expressed in a way that generators can be constructed and maintained.

The current generator model is shown in Figure 2. One generator is constructed to produce several members of a family. All possible variabilities are handled by the generator. So, for example, if there is no requirement for reentrancy, no reentrant specific code is produced. However, the generator has several places where decisions about whether to generate reentrant code must be made.

A system for creating domain specific languages (DSLs) to describe the components and a generator framework are being explored. The framework allows one to compose an appropriate generator given a specification for a family of components. In such a system, generators for a domain are created by composing them from compatible parts. Each generator in the family of generators works on an unique set of variabilities.

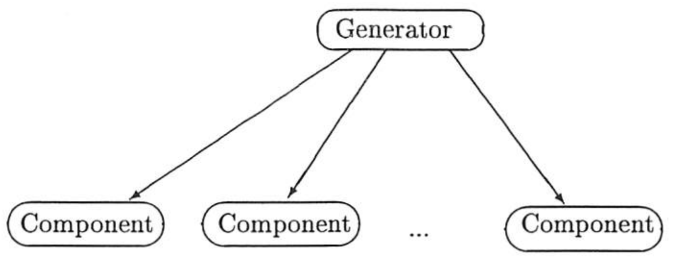


Figure 2: Current generator model.

Figure 3 shows a model for a family of generators. Heavy arrows show generator extension (i.e., sharing of variabilities). In the example, G_4 is an extension of G_2 and G_6 is a shared extension of G_3 and G_5.

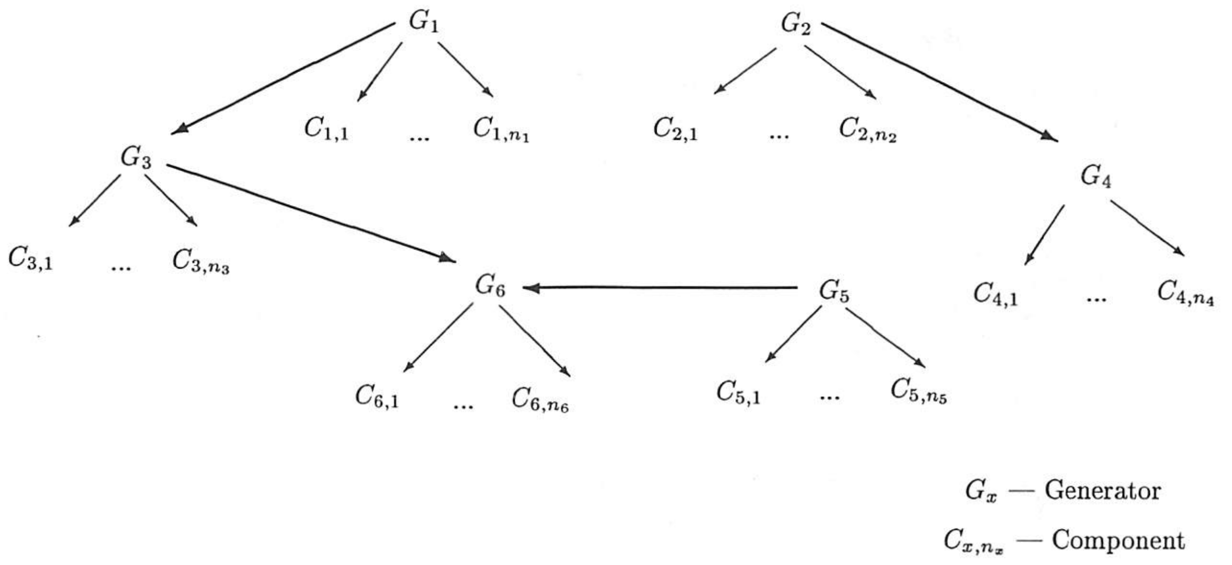

G_1
$C_{1,1}$ ... C_{1,n_1}
G_2
$C_{2,1}$ ... C_{2,n_2}
G_3
$C_{3,1}$ ... C_{3,n_3}
G_4
$C_{4,1}$ ... C_{4,n_4}
G_5
$C_{5,1}$ ... C_{5,n_5}
G_6
$C_{6,1}$ ... C_{6,n_6}
G_x — Generator
C_{x,n_x} — Component

Figure 3: A family of generators.

The desired system allows the generator producer to work at a high level, using language appropriate for the domain. The system performs the necessary semantic analysis and bookkeeping to construct generators from the appropriate parts. Each generator functions like the generators described in this paper. At least one VHLL, possibly more, is used in the system. The generators are written in a VHLL. The framework for the system will most likely be written in a VHLL as well.

6 Availability

The original **scangen** sources may be obtained as a **shar** file from the author. These are the source files used in the M.S. thesis, which may also be obtained from the author as a compressed, encoded PostScript file. The author may be contacted at pollice@centerline.com or gpollice@cs.uml.edu.

7 Acknowledgments

I would like to thank CenterLine Software, Inc. for providing hardware and software resources used to develop and refine the **scangen** generators. Thanks also go to Bill McKeeman and David Weiss for providing feedback on the work. Their comments have and continue to shape the direction of my work.

8 Biographical Information

Gary F. Pollice is a software engineer at CenterLine Software, Inc., Cambridge, MA. He is also in the doctoral program at the University of Massachusetts, Lowell. He received a B.A. in mathematics from Rutgers, the State University of New Jersey, and a M.S. in computer science from U. Massachusetts, Lowell. His research interests are software reuse, compiler technology, and software engineering.

References

[1] Reuse-driven software process guidebook. Technical Report SPC-92019-CMC, Software Productivity Consortium, Herndon, VA, November 1993.

[2] The SDDR design concept. Mosaic pages, 1994.

[3] American National Standards Institute. *American National Standard for Information Systems — Programming Language C*, February 1990. Doc. #X3J11/90-013.

[4] Jon Bentley. Template-driven programming. *Unix Review*, 12(4):79–88, April 1994.

[5] J. Craig Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.

[6] Kyo C. Kang, Sholom Cohen, Robert Holibaugh, James Perry, and A. Spencer Petersoft. A reuse-based software development methodology. Technical Report CMU/SEI-92-SR-22, Software Engineering Institute, Pittsburgh, PA 15213, January 1992.

[7] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[8] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

[9] W. M. McKeeman. Personal Correspondence, 1993. Electronic mail message.

[10] John K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, Reading, MA, 1994.

[11] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[12] Gary F. Pollice. Component generation as a software reuse technique illustrated with a C scanner generator. Master's thesis, U. Massachusetts, Lowell, 1994.

[13] Christopher J. Van Wyk. AWK as glue for programs. *Software—Practice and Experience*, 16(4):369–388, April 1986.

[14] Larry Wall and Randal L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

Dixie Language and Interpreter Issues

R. Stockton Gaines
U. S. C. Information Sciences Institute
4674 Admiralty Way
Marina del Rey CA 90292
gaines@isi.edu

Abstract

Dixie (Distributed Internet Execution Environment) provides a base for sending programs called Dixie applications to Internet sites for execution. It provides the features generally found in operating systems, such as a file system, multiprocessing, interprocess communications, etc., and in addition capabilities to permit Dixie applications to interact with resources at the local site. Security is of first importance; it must not be possible for a Dixie application to have an undesired effect on the local system. This paper explains the Dixie concept and discusses language and execution issues. The languages understood by Dixie, at least initially, will fall in the class of Very High Level Languages, not the least because these languages will support the security requirements of Dixie, as well as the command language requirements. Dixie complements these languages, and provides uniform platform independent of the local hardware and operating systems to support Dixie application programs.

Introduction

Dixie (Distributed Internet Execution Environment) is a virtual operating system and program execution environment that is portable. Once installed on a system connected to the Internet (an Internet host, or simply "host", here), it can execute programs in the languages it supports. Dixie, therefore, provides a means of sending a program to other Internet hosts for execution. Programs which execute in the Dixie environment are called Dixie applications. They are an instantiation of the concept of knowbots or intelligent agents that can travel through the Internet carrying out useful actions.

Dixie complements the recent work on very high level programming languages by providing an interface that is consistent across systems, is secure in that programs executing on Dixie are harmless to the local host system, and provides full operating system functionality. A program written in a language embedded in Dixie will see the same operating system interface anywhere in the Internet, independent of either the operating system or the compilers of the host system itself.

Dixie solves another problem, that of software portability. A Dixie application can be run on any host with Dixie installed, without need for changes to be compatible with the host's underlying operating system, compilers or hardware. Dixie therefore is a vehicle for software distribution. Furthermore, because Dixie installations are themselves accessible through the Internet, a natural means of remote maintenance of Dixie applications (as well as Dixie itself) is available.

Whereas portable servers such as World Wide Web (WWW) are primarily of interest on hosts that act as servers at Internet sites, Dixie will also be useful when installed on workstations. Dixie will include a GUI package so that a Dixie application running on a workstation can be interactive with the workstation user.

Dixie includes a full operating system model, called the Dixie Host Interface (DHI). The main features are: processes, including multiple processes in execution, shared memory between processes, interprocess communications, and process scheduling and swapping; a complete file system interface; and device and service interfaces to the host system on which it is installed.

The first priority in designing and implementing Dixie is that it provide a secure execution environment.

There are two types of security of concern, both important. First, a Dixie application must not be able to harm the host system on which it runs in any way. Second, communications to and from both Dixie applications and the DHI must be authenticated, and be reliable (that is, the message received must be verifiably the message sent). These security considerations will not be discussed further here, but dominate the design of Dixie.

Another system that offers the ability to send a program to a remote site for execution is Safe-Tcl [1, 7, 8]. Safe-Tcl is the most recent version of an active mail system (termed enabled mail in Safe-Tcl papers). Enabled mail is mail that when read by the receiver, executes as a program. Dixie takes a much broader view of the issues and requirements for executing programs at remote Internet sites, but many of the security issues are similar. A good discussion can be found in the Safe-Tcl references.

The Dixie Host Interface will support multiple program execution environments. Initially these will be based on interpreters. Again, the goal is to provide a host-independent environment that supports useful languages.

Dixie will combine three important existing components: the Prospero file system [5], the Tk [6] GUI package, and interpreters for several languages. At the time of writing it is expected that Python [9] will be of great interest. This language is an unusual combination of elegance, simplicity and power, with a number of features that are particularly suitable for Dixie (especially its form of modules). Tcl [6], already in widespread use, is another powerful and important language that will be integrated into Dixie. Other languages of interest are Perl [10], REXX [3] and, when and if it becomes available, Telescript from General Magic. All of these languages are implemented as interpreters, and are suitable both as command languages and programming languages. Since in many cases a Dixie application will execute on a remote site from the invoker of the program, and the invoker will not be connected in a session with the application, the ability of the application to generate commands to its operating system (DHI) as well as lower level operating system calls is an important virtue of all these languages.

One motivation for Dixie is to provide a method that permits programmatic access to local resources through the Internet in a safe way. For example, sites that maintain databases may wish to make the information in the database accessible without exporting the entire database. An SQL interface will be incorporated in DHI through which accesses to local databases can be defined and controlled by the host owner. For example, if various Departments of Motor Vehicles are interested in making information about automobiles and accidents available, but prohibiting access to any personal information about drivers or automobile owners, appropriate views of the database can be defined that will not support the retrieval of such prohibited information. A Dixie application running on such a host can issue SQL commands against these views, but cannot otherwise access the database.

The file system interface will be based on Prospero [5]. It is already being used extensively, and has, for example, been used as the basis for the archie server. Prospero defines a mapped view of the underlying file system. The view that is presented through Prospero consists of a set of directories and files that may be different from the actual structure of the file system of the host computer. The mapping will be definable by the owner of the host system. (Prospero also includes the ability to make visible non-local files that reside on other systems. This, too, may be valuable for Dixie).

An important aspect of Prospero is that attributes can be associated with each Prospero visible file and directory. These attributes can include access methods. For example, a read access method can be defined for each file. When the file is accessed, the routine specified for the file is invoked, rather than simply reading the file in the normal manner provided by the host file system. The attributes can include additional security mechanisms. One example would be the association of an access control list with a file, designating on a per file basis the rights of specific authenticated individuals. Attributes associated with directories would include the right to create a file, and to designate its type. For example, it would be possible, and useful, to restrict the creation of files to files that can be read but not executed. Dixie, through the use of Prospero, will be able to insure that no Dixie application can install a file in the local file system that is executable, which will prevent many well known attacks on systems. The power to control exactly how the Dixie applications can interact with the local file system, including which portion of it is visible, and through the use of file and directory attributes place additional limitations on the access to the file system and the ways in which files are created, named or renamed and modified leads to a high degree of security.

Prospero provides the ability to map a single file into an entire file system, from the viewpoint of a Dixie application. Prospero can also map a disk partition as a file system. This will isolate it completely from the host file system, if that is desirable. As can be seen from these examples, Prospero provides complete flexibility in providing persistent storage through a file system interfaced for Dixie and Dixie applications, with the ability to expose those parts of the host file system that the host owner desires, while restricting all other accesses.

In general, restrictions on the use of the host system's resources will be implemented within DHI. Since DHI provides all support for Dixie applications, which cannot invoke the host operating system directly, restrictions on the language itself will be minimized.

For reasons of efficiency or functionality, it may be desirable that a Dixie application be able to make calls on routines that are compiled to run directly on the host computer. For example, if Dixie had been available and in widespread use, it could have been used as the basis for finding the largest prime number using many computers throughout the world. The heart of this distributed application was a relatively simple C program. All of the communications and coordination parts of the application could have been handled through a Dixie program for each host, since the computation requirements for these were not great. But it would have been necessary to provide an interface to the C subroutine from a Dixie application.

The main issue here is security. The host owner must be able to trust the C program. The host owner could trust the program if written locally, or obtained from a reliable source. Trust could also be based on an inspection of the program's source code, for programs that are simple enough. In the example just given, this could be straightforward. The program should inspect its inputs to insure that they are valid, should not make any system calls, and should communicate with the DHI in a straightforward way, such as accepting a single value as an argument and returning a single value. The routine would need to be registered by the host owner as callable through the DHI in order to be accessible to a Dixie application. To deal with more than very simple cases may be a research question.

Language and Interpreter Issues

The philosophy that motivates Dixie is that there is a clear distinction between an operating system and a programming language. Far too much of the operating system tends to get built into programming languages, limiting flexibility and applicability. This philosophy suggests that the abstractions presented to the programmer should be at a high enough level that there is freedom to do what makes sense during code generation, program execution and in the operating systems to deal with issues of memory management, process structuring and scheduling, etc.

A process has an internal behavior and an external behavior. The programming language provides mechanisms for defining objects that populate the internal environment and specifying actions on those objects. A language is also needed to describe the external actions of a process, but that language is, according to the philosophy being espoused here, not part of the programming language. Rather it is a language invoked through the programming language by calls to routines that cause external actions, and by emitting statements in a language that is understood external to the process. A great virtue of many very high level languages is that they provide good tools for generating these statements for external consumption.

An example from ADA may help to illustrate the point. ADA includes as language constructs "fork" and "join". Fork and join are process management actions. By including these as primitives, ADA was forced to add a lot more baggage within the language to define and manage what amounts to pseudo processes. These features in turn impose restrictions on the operating system, or else result in a complicated run time package to support ADA. If fork and join are calls on routines that are supplied separately from the programming language, they can have a semantics suitable to both the operating system and hardware environment in which the program will execute, and can be optimized for the needs of different types of applications.

The separation of concepts between the programming environment and the supporting operating system environment of Dixie leads to a smaller set of requirements for the languages that provide the execution environment for Dixie applications. The required functionality, to the extent possible, will be provided by a set of run time callable routines that are common to all the programming environments. This has the additional virtue that Dixie can evolve without the need to change all the language interpreters when there is a change in the DHI.

Since the Dixie Host Interface acts as the operating system for a set of Dixie processes that are executing Dixie applications, it must provide for the synchronization of the activities of these processes. A design objective of Dixie is to develop a set of synchronization and coordination tools that will support both processes running on the same machine and processes that are distributed among multiple machines. Semaphores and other synchronization mechanisms will be built into the DHI. Such tools are not ordinarily included in operating systems, but there are a couple of advantages. First, they can be made simple and efficient. In addition, the scheduling and swapping policies for Dixie processes can be aware of process synchronization activities, also improving efficiency.

An issue that has not received much attention from the programming language community is how a programmer can view and act on a program from within the program. At least two aspects of this are pertinent to Dixie applications. Dixie applications will often execute far away from their creator, and must be able to deal with the local environment in ways anticipated by the programmer, but not interactively with the programmer or invoker during execution.

One aspect of making a program aware of itself is to make accessible to the program the attributes of objects within the program. These attributes are known to the compiler or interpreter, and often to the run-time code, but generally are not accessible by the program itself. Objects (simple variables, arrays, structures, procedures and functions, etc.) have attributes such as type, dimension, and whether or not they have been written to (set). There are times when a programmer would like to obtain the values of these attributes. Variables to hold these values can be created and set explicitly in some circumstances, but not always. For example, when an array is passed by name, it would often be convenient to obtain the size of the array from attributes known to the run-time code.

The current type of a variable is an interesting case in several very high level languages. In some of these languages, the type of all variables is "string" at the language level, but has a dynamic type such as integer, floating point or string at run time. Though the interpreter knows or can determine this dynamic type, it is not always available to the programmer. As an example of its use, one might like to construct a sort routine that checks on the types of the elements being sorted, and acted according to this information.

Another aspect of a program that is likely to be of interest for Dixie applications is how long the program has run, according to some measure. Host computers that run Dixie so that Dixie applications can access local resources may wish to provide limitations on the amount of execution time any one Dixie application can consume. An approximation for this is the number of statements executed. The programmer may wish to write a Dixie application that uses most of the available time, and then interrupts itself to prepare a message reporting the results obtained before terminating (or being terminated). Methods of making this information available conveniently will be explored.

A second area of interest is how one constructs programs to react to errors. The REXX language has incorporated the ON CONDITION concept from PL/1. This is very useful in many cases. The basic concept is that if a certain condition arises during a program, this creates a "trap" to a specified subroutine. It may or may not be possible to return to the point at which the trap occurred, depending on the cause of the trap and details the programming language.

This notion of "if some state is reached, invoke this action" as a global statement to be checked for continuously during program execution, in contrast to explicitly programmed checks, is very powerful. It leads to a very useful kind of internal multithreading within programs. I refer to this as "internal" because it is not visible to the operating system.

REXX includes the ability to turn condition checking on and off for specific events. It is very useful for building routines that can react to errors in dealing with the operating system without placing lots of messy error checking code in the middle of what may be already complicated blocks of code.

There are several issue in implementing and using an on condition feature. Obviously it can be expensive to carry out checks continuously, so this must be dealt with in sensible ways. It must be clear to the programmer who cares what the overhead is in using this feature. It must be possible to turn checking on and off, so that sections of code where the condition being checked for will not occur need not bear the overhead.

When a trap occurs, the question arises of how to determine where the trap was generated. It would be nice to be able to insert labels in the code for this purpose (as it would be for some debugging tools). If this were possible, the value of a variable associated with the trap could be checked to identify the trap location. In REXX it is possible to obtain a line number, which is useful for post-mortem debugging. This is hard to make use of at run time because line numbers will change each time the program is modified, and it is a problem to keep track of them accurately for use within a trap routine.

The availability of an on condition that is triggered by the number of statements executed would be a useful solution to the problem mentioned above of trapping near the end of a Dixie applications allotment of execution time.

References

[1] N. Borenstein and M. Rose, "EMail with a Mind of its Own: the Safe-Tcl Language for Enabled Mail", to be published in *ULPAA `94.*

[2] B. Borden. R. S. Gaines and N. Shapiro, "MH, A Message Handling System for the UNIX Operating System", The Rand Corporation, R-2376-PAF, October 1979.

[3] M. F. Cowlishaw, *The REXX Programming Language*, Prentice Hall, 1990.

[4] R. S. Gaines, "An Operating System Based on the Concept of a Supervisory Computer", *Communications of the ACM*, Vol.15, No.3, March 1972.

[5] B. C. Neuman, "The Prospero File System: A global file system based on the Virtual System Model," *Computing Systems,* 5(4),p. 407-432, FAll 1992

[6] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading Massachusetts, 1994.

[7] M. Rose and N. Borenstein, "A Model for Enabled Mail (EM)", draft in preparation.

[8] M. Rose and N. Borenstein, "MIME Extensions for Mail- Enabled Applications: Application/Safe-Tcl and Multipart/enabled-mail", draft in preparation.

[9] G. van Rossum, Python 1.0.1, documentation and code available for anonymous ftp from ftp.uu.net in /

languages/python.

[10] L. Wall and R. Schwartz, *Programming Perl*, O'Reilly & Associates, 1990.

Stockton Gaines has worked in the areas of computer operating systems and computer security for over 25 years. His paper "An Operating System Based on the Concept of a Supervisory Computer" [4] was presented at the 3rd Symposium on Operating Systems Principles in 1971. He was chairman of the ACM's Special Interest Group on Operating Systems (SIGOPS) and Operating Systems editor of the Communications of the ACM from 1975 through 1980. He was a consultant starting in 1981, and consulted on operating systems for IBM, Honeywell and Control Data Corporation, among others, during the years 1981-1989. Dr. Gaines directed ISI's research on parallel computing from 1989 to 1992, which include the porting of the Mach operating system to a distributed memory parallel computer . He developed the concepts of System Manager and Job Manager which form the basis of the Prospero Resource Manager being developed by Cliff Neuman at ISI.

Dr. Gaines chaired the first conference on computer security, held in Princeton, NJ in 1972. He chaired the technical committee to oversee the development of a secure version of Unix for ARPA during 1976-1977. Together with Norman Shapiro of the Rand Corporation, he designed the MH mail handling system [2], and he directed its development. Subsequently, he did research on secure message system . As part of his consulting he worked on security issues for a number of clients.

Feature-Based Portability

Glenn S. Fowler (gsf@research.att.com)
David G. Korn (dgk@research.att.com)
John J. Snyder (jjs@research.att.com)
Kiem-Phong Vo (kpv@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

Current computing platforms encompass a dizzying variety of hardware and software. A software application may live or die based on how portable it is. Much has been written and talked about how to enhance portability. But few tools are available to support writing portable code and, more importantly, to encode porting knowledge. This paper describes IFFE, a tool and an accompanying programming style that supports software portability. IFFE has enabled the porting and construction of many large software applications on heterogeneous platforms with virtually no user intervention.

1. Introduction

Over the past 10 years our department at AT&T Bell Laboratories has been engaging in writing a number of popular software tools and libraries. Some examples are KSH [BK89], a shell language, NMAKE [Fow85], a language and system to build running code from source, EASEL [Vo90, FSV94], a language and system to build end-user applications, and *sfio* [KV91], a library for buffered I/O. These tools and libraries critically depend on various resources provided in the underlying platforms. A major problem is that such platform resources are not always available or usable in the same form. For example, to tell whether or not a file descriptor is ready for I/O, on a BSD-derived system, one should use the `select()` system call while on newer System V systems, `poll()` is required. The problem is exacerbated by the fact that there are many hybrid systems, some from the same vendor, that provide mixed services. In a different direction, most systems come with standard libraries such as string and mathematical packages but their implementations vary in quality. An extreme case is the VAX family of machines that come with hardware instructions for certain string and character look-up operations that are more efficient than any handcrafted software. It is desirable to take advantage of such platform-specific features to optimize the software. Of course, in all cases, we have to be certain that a platform feature used will work as expected. In sum, the porting problem is this: how can we certify that a particular feature exists on a particular software/hardware platform and that it does what is required?

This paper describes a tool IFFE (IF Features Exist) and an accompanying programming style to help with writing portable code and gives a brief comparison of IFFE to other approaches. IFFE has enabled us to: (1) port software to new platforms with minimal changes, (2) codify learned knowledge during porting, (3) apply such knowledge without relying on users to specify software/hardware parameters at each installation, and, last but not least, (4) take advantages of special platform features to tune for performance.

2. A programming style for portability

Our overall approach to portability is to program applications against high level libraries that hide differences among underlying platforms. Then porting effort is mostly confined to the library code. The traditional approach for selecting different code variants is to use "`#ifdef` *selector*" where *selector* is a predetermined symbol based on some broad categorization of machine type (e.g., `sun` or `sgi`) or operating system type (e.g., `BSD` or `SYSV`). Such a broad categorization is convenient and does work in limited cases. It is also necessary because the specific value of *selector* is typically supplied by some user during a build and most users neither know nor have the means to find out and evaluate alternatives in the full set of locally available features. However, in modern environments where mixtures of services are typical, more often than not this traditional way of code selection will miss the mark and lead to the construction of bad code.

We solve the porting problem by applying a programming style supported by IFFE. It is best to show this with an example. Consider the following code fragment taken from the source of the `sfpopen()` function of the *sfio* library:

```
1:  #include      "FEATURE/vfork"
2:  #if _lib_vfork
3:  #    define fork vfork
4:  #    if _hdr_vfork
5:  #         include      <vfork.h>
6:  #    endif
7:  #endif
```

Line 1 includes a file `FEATURE/vfork` that defines two symbols, `_lib_vfork` and `_hdr_vfork`. Line 2 tests `_lib_vfork` for the existence of the system call `vfork()`. Line 3 redefines `fork()` to `vfork()` if it exists. `sfopen()` uses `fork()` to create a child process which, after some minor processing, will be overlaid by a new command. So redefining `fork()` as `vfork()` is good as the latter does the same job without the expensive operation of copying all data of the parent. This use of `vfork()` works fine except on a SUN SPARC which has a major problem that registers modified by a child process get propagated back to its parent. SUN provides a compiler directive in the header file `vfork.h` to generate code that avoids this bug. The existence of this file is tested on line 4 and its inclusion is done on line 5. It is worth emphasizing that even though this problem is currently known to occur only on a SUN SPARC, its solution is targeted at the nature of the problem and not at the machine. Further, the solution is encoded in a form independent of machine architectures.

To complete the example, we need to see how the file `FEATURE/vfork` containing the symbols `_lib_vfork` and `_hdr_vfork` can be correctly and automatically generated. This is done via the IFFE language and system. The keen reader may have noticed that a subdirectory `FEATURE` is used to store the header file `vfork` which contains the definitions of the required tokens. This file is generated from a specification file `vfork` in a parallel directory `features`. The content of `features/vfork` is:

```
lib vfork
hdr vfork
```

The line "`lib vfork`" determines if `vfork()` is a function in some standard library (e.g., `/lib/libc.a`) by generating, compiling and linking a small test program that contains a `vfork()` call. Similarly, the line "`hdr vfork`" determines if the header file `vfork.h` exists by compiling a small program containing the line "`#include <vfork.h>`". Below is the output file `FEATURE/vfork` for a SUN SPARC. Note that to prevent errors with multiply inclusions the generated symbols are automatically wrapped with the wrapper `#ifndef` and `#endif` which is generated from the base name of the feature test file `features/vfork` and `sfio`, the parent directory of `features` and package name. By the way, in case the reader may wonder why `FEATURE` does not have an `S` to parallel `features`, this is done to distinguish the two directories on operating systems such as Windows NT where cases are indistinguishable in directory and file names.

```
#ifndef _def_vfork_sfio
#define _def_vfork   1
#define _lib_vfork   1    /* vfork() in default lib(s) */
#define _hdr_vfork   1    /* #include <vfork.h> ok     */
#endif
```

Though the above example works, we are actually a little too trusting as compilability is not equivalent to execution correctness. For complete safety, IFFE scripts can specify programs that must compile, link and execute successfully. Below is another IFFE specification from *sfio* that tests for the correct register layout of a given VAX compiler:

```
vax asm note{ standard vax register layout }end execute{
    main()
    {
    #ifndef vax
        return absurd = 1;
    #else
        register int    r11, r10, r9;
        if(sizeof(int) != sizeof(char*))
            return 1;
        r11 = r10 = r9 = -1;
        asm("clrw    r11");
        if(r11 != 0 || r10 != -1 || r9 != -1)
            return 1;
        asm("clrw    r10");
        if(r11 != 0 || r10 != 0 || r9 != -1)
            return 1;
        asm("clrw    r9");
        if(r11 != 0 || r10 != 0 || r9 != 0)
            return 1;
        return 0;
    #endif
    }
}end
```

The above code will compile and run correctly only on a VAX with a proper compiler. If that is the case, the output would be as below and we would know that the register layout is as expected so that certain hardware instructions can be used safely for optimization.

```
        #define _vax_asm 1   /* standard vax register layout */
```

IFFE specifications can be integrated with `makefiles` in the obvious fashion. Users of the NMAKE system for code construction enjoy this integration automatically since NMAKE scans the source code for any implicit header file prerequisites (a.k.a. `#include` dependencies) including the `FEATURE` files. The additional NMAKE metarule shown below provides the action to generate the `FEATURE` files. Note that where old MAKE is still used, NMAKE can also be used to generate `makefiles` that contain all such header dependencies.

```
        FEATURE/% : features/% .SCAN.c (IFFE) (IFFEFLAGS)
            $(IFFE) $(IFFEFLAGS) run $(>)
```

To summarize, the programming style that we adhere to is:

1. Determine needed features that may have platform specific implementations.
2. Write IFFE probes to determine the availability and correctness of such features.
3. Instrument `makefiles` to run such IFFE scripts and create header files with properly defined configuration parameters.
4. Instrument C source code to include `FEATURE` header files and use `#define` symbols in these files to select code variants.
5. Restrain `FEATURE` file proliferation by limiting their use to libraries when possible.

By following the above steps during any port of a software system, porting knowledge is never forgotten. Indeed, such knowledge is coded in a form that is readily reusable in different software systems. In extreme cases `FEATURE` files generated on one platform may be used to bootstrap software on another. We have used this technique to port much of our software to Windows NT. The port started with `FEATURE` files from a mostly ANSI/POSIX system which were edited as necessary until `ksh` was up and running. Then, other software systems could be rebuilt with IFFE whose interpreter is written in the Bourne shell language [Bou78].

3. The IFFE language

An IFFE input file consists of a sequence of statements that define comments, options or probes. A comment statement starts with # and is ignored. An option statement is used to customize the execution behavior of the IFFE interpreter such as changing the compiler or resetting debugging level. The heart of the IFFE language is the probe statement. Below is its general form:

```
        type name [ header ... ] [ library ... ] [ block ... ]
```

Here, *type* names the type of probe to apply, *name* names the object on which the probe is applied, *header* and *library* are optional comma-separated lists of headers and libraries to be passed to the compiler (non-existent ones are ignored), and *block* are optional multi-line blocks that define the probe's code.

type and *name* may be comma-separated lists in which case all *type*s are applied to all *name*s. Though *type* can be any value defined by users, probes for certain common types are provided by default. Below is a partial list of the common types.

- `lib`: Checks if *name* is a function in the standard libraries.
- `hdr`: Checks if `#include` <*name*`.h`> is valid.
- `sys`: Checks if `#include` <`sys/`*name*`.h`> is valid.
- `key`: Checks if *name* is a C language keyword.
- `mac`: Checks if *name* is a C preprocessor macro.
- `typ`: Checks if *name* is a type defined in `sys/types.h`, `stdlib.h` or `stddef.h`.
- `cmd`: Checks if *name* is an executable in a standard directories such as `/bin` or `/etc`. If the command is found, the symbol `_cmd_`*name* is defined. In addition, each directory containing the command generates the symbol `_cmd_`*dir_name*.

The default output for a successful probe is shown below. Note that since the constructed output symbols must contain *type*, the names of the above default types are made short so that there is less chance of name conflicts in older compilers that restrict symbols to less than 8 characters.

```
#define _type_name 1   /* comment */
```

For example, the probe statement "`lib bcopy,memcpy`" checks to see if `bcopy()` and/or `memcpy()` are available in a standard library (most likely `/lib/libc.a`). On an old BSD Unix system, the output of this probe is likely to be:

```
#define _lib_bcopy 1   /* bcopy() in default lib(s) */
```

If the application code desires to use `memcpy()` exclusively then the probe output can be used to mimic or replace `memcpy ()` as follows:

```
#if _lib_bcopy && !_lib_memcpy
#define memcpy(to,from,size) (bcopy(from,to,size),to)
#endif
```

The optional *block*s in a probe statement are labeled and indicate actions to be done. Each block is of the form:

```
label{
    line
    ...
}end
```

Certain block labels indicate that the respective blocks contain actions to be done after a probe is executed. These labels are:

`fail`: If the probe fails then the block is evaluated as a shell script and its output is copied to the output file.

`pass`: If the probe succeeds then the default output is suppressed, the block is evaluated as a shell script and its output is copied to the output file.

`note`: If the probe succeeds then the block is output as a single comment.

`cat`: If the probe succeeds then the block is copied to the output file.

Other block labels mean that the respective blocks define probe code to override the respective default code templates if any. A probe is consider successful if it exits with status 0. These block labels are:

`run`: The block is run a shell script and the output is copied to the output file.

`preprocess`: The block is preprocessed as a C program.

`compile`: The block is compiled as a C program.

`link`: The block is compiled and linked as a C program.

`execute`: The block is compiled and linked as a C program and is then executed; the output is ignored.

`output`: The block is compiled and linked as a C program, is then executed and the output is copied to the output file.

Below is an example of checking to see if the `mmap()` system call is available and if it does the job. In this case, the default probe code template to check for the existence of `mmap()` in some standard library (e.g., `/lib/libc.a`) is not good enough. The **`execute`** block indicates that the given program must be compiled and run to ensure that `mmap()` exists and works as expected. The probe success is defined by returning `0` at the end of execution.

```
lib mmap sys/types.h fcntl.h sys/mman.h execute{
    main(argc,argv)
    int   argc;
    char* argv[];
    {   int     fd;
        caddr_t p;
        if((fd = open(argv[0],0)) < 0)
            return 1;
        if(!(p = (caddr_t)mmap(0,1024,PROT_READ,MAP_SHARED,fd,0L)) ||
           p == ((caddr_t)-1) )
            return 1;
        return 0;
    }
}end
```

4. Writing and executing probes

A typical probe is a fragment of C code to be processed in some form. As discussed in Section 3, certain probe types come with default code templates but others must be supplied. This section discusses the style for writing C code in probe tests and briefly talks about the IFFE interpreter.

4.1. C code in IFFE probes

To eliminate duplication and ease the writing of probe code, IFFE automatically provides a number of preprocessor macros for code in the blocks (`preprocess`, `compile`, `link`, and `execute`). With proper use of these macros, code for probes can be written to be transparently compilable with different C language variants including K&R-C, ANSI-C, and C++ . The macros are:

- `_STD_`: This symbol is `#defined` to `1` if the compiler is some flavor of ANSI-C or C++. Otherwise, it is defined to be `0`.
- `_VOID_`: This is defined to be `void` for ANSI-C and C++ and `char` for older C.
- `_ARG_((`*x*`))`: This macro function expands function prototypes depending on the underlying C language. Note that the extra pair of parentheses is required to avoid variable argument macro conflicts.
- `_NIL_(`*type*`)`: This is a convenient macro that expands to `((`*type*`)0)`.
- `_BEGIN_EXTERNS_`, `_END_EXTERNS_`: These macros should be used around `extern` declarations to prevent their C names from being mangled by certain C++ implementations.

Below is another example probe taken from the *sfio* library. This probe checks to see if the routine `_cleanup()` of the *stdio* package is called when a program exits. *sfio* uses this information along with other information on exiting conventions of the local environment to configure its own clean-up procedure upon program exiting. Note that the type `exit` is application-defined.

```
exit cleanup note{ exit() calls _cleanup() }end execute{
    _BEGIN_EXTERNS_
    extern void exit _ARG_((int));
    extern void _exit _ARG_((int));
    _END_EXTERNS_
    void _cleanup() { _exit(0); }
    main() { exit(1); }
    }end
```

The probe works by explicitly calling from `main()` the `exit()` routine with an exit status `1` to signify probe failure. However, if `_cleanup()` is called implicitly, it will call `_exit()` which causes the program to exit with status `0` to signify probe success. Note that by necessity this probe must be both compiled and run. `_exit_cleanup` is defined on our local

SunOS 4.1 system. We shall not go into detail about why this probe is necessary. Suffice it to say that it was done when the *sfio* library was ported to an environment where there is no `atexit()`-like function and the code is expected to compile in a normal C environment but it may be linked with C++ code.

4.2. The IFFE interpreter

As IFFE is a part of the build procedure, it must be maximally portable. For this reason, the IFFE interpreter is written in the Bourne shell language which is supported on all known UNIX systems. It currently stands at about 1200 lines of code. The interpreter has a single option: if the first argument is "–" then the probe output is written to the standard output rather than the default `FEATURE/`*name*. Other arguments are interpreted as IFFE statements, where a ":" argument is the statement separator.

Below are a few typical interpreter invocations. The first one runs the probe tests in `features/lib`. The second one sets the compiler to `CC`, i.e., the C++ compiler, then runs the probe tests in `features/stdio.c`. The last one tests to see if `socket()` and `sys/socket.h` are available and writes the output to the terminal. This is a useful way to find out quickly certain information about the programming environment.

```
iffe run features/lib
iffe set cc CC : run features/stdio.c
iffe - lib,sys socket
```

5. Comparisons with other approaches

The idea of automatically configuring a software system by probing the native platform underlies the build procedure of many popular systems such as PERL [WS90] and older versions of KSH and NMAKE. Some have gone as far as writing code to make exhaustive lists of virtually all machine properties [Pem92]. The problem with this approach is that the lists often include much more than necessary and may cause unwanted side effects due to the large number of symbols. The programs that generate such lists are also susceptible to the usual problems in porting and maintenance. Another approach based on a combination of parameter and configuration files is reported in [TC92]. A cursory look at the examples given in the paper show that this approach relies on some scheme of broad platform classification (e.g., references to `bsd43` or `mips`). As observed in Section 2, this scheme does not always work and requires much more knowledge from the installer than necessary.

Closer in spirit to IFFE is the METACONFIG system by Larry Wall. This works by maintaining a glossary of symbols and a depository of probe units corresponding to the symbols in the glossary. Then, each application generates a shell script that includes all probes corresponding to source symbols that appear in the glossary. This script is packaged with the build procedure and run to generate the correct definitions of needed symbols each time the system is rebuilt. Though this is similar to the IFFE approach, there are fundamental differences between the systems at different usage levels as discussed below.

At the specification level, the symbols in the METACONFIG glossary are similar to those generated by IFFE from the *type* and *name* attributes of probe statements. However, because the glossary is centrally maintained, it can become arbitrarily large, cumbersome and difficult to master. By separating *type* and *name* as independent components of a symbol (Section 3) and defining a small number of default *type* probe code templates, IFFE reduces the effort to specify probes for such symbols. In fact, most IFFE probes for common objects (e.g., checking the existence of `bcopy()`) reduce to single lines of specification. Sharing of porting knowledge is done by reusing probe scripts. Thus, each application can define or acquire exactly the symbols and probes that it needs.

At the code shipment level, METACONFIG introduces an asymmetry between the provider and receiver of a software system because the receiver may lack the means to make the METACONFIG-generated script. If new probes are required during a port by the receiver, the only recourse is to modify this script which is not a simple procedure since the script can be quite large and complex. Further, since the script is automatically generated, it is also the wrong place to place such porting knowledge. IFFE scripts are treated as parts of the source code. The IFFE interpreter is usually shipped along with the source code. In this way, both the provider and receiver of a system see exactly the same thing. As the IFFE language is relatively simple, new probes required by porting can be easily recorded.

Finally, at the development level, the reliance on a single script for parametrization means that slight changes in probes may trigger long rebuilds since this script must be rebuilt and run. Though this is not a problem with software already advanced to a stable stage, it can be a serious nuisance during porting efforts. In addition, in an environment where parallel build is available (e.g., running `nmake` on a network of homogeneous machines), the METACONFIG-generated script can be the bottleneck and cause ineffective use of computing resource. The IFFE approach of making a correspondence between source probe files (in `features`) and generated headers (in `FEATURE`) holds an advantage because separate headers can be generated on a as-needed basis and then they can be simultanously generated on different processors if the environment allows. This approach also fits well with the general philosophy of build tools such as MAKE [Fel79] and NMAKE that certain objects are generated from other source objects.

6. Conclusion

As stated at the start of this paper, the portability problem boils down to finding out from a platform exactly which of its features are required and whether such features perform as expected. Any scheme of answering this question based on a broad classification of platforms (e.g., BSD vs. SYSV or SPARC vs. MIPS) is doomed to fail because modern environments tend to contain ad hoc mixtures of features. Even when a required feature is available, a bane to programmers is that its implementation quality can vary greatly from platform to platform. For example, the `mmap()` system call is a good alternative to `read()` for reading disk data on many modern UNIX systems because it avoids a buffer copy. But on certain platforms, `mmap()` simply does not work and on others, its performance can be worse than `read()`. This makes it hard to effect high quality implementation of critical software components such as the buffered I/O library *sfio*. IFFE and its accompanying programming style provide an effective solution by enabling programmers to target specific platform features and perform a variety of tests to determine their acceptability. The high level IFFE language also provides a convenient mechanism to record porting knowledge in a form that is easily shared among software developers.

7. References

[BK89] Morris Bolsky and David G. Korn. *The KornShell Command and Programming Language.* Prentice-Hall Inc., 1989.

[Bou78] S. R. Bourne. The Unix Shell. *AT&T Bell Laboratories Technical Journal,* 57(6):1971–1990, July 1978.

[Fel79] S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience,* 9(4):256–265, April 1979.

[Fow85] Glenn S. Fowler. The Fourth Generation Make. In *Proceedings of the USENIX 1985 Summer Conference,* pages 159–174, June 1985.

[FSV94] Glenn S. Fowler, John J. Snyder, and Kiem-Phong Vo. End-User Systems, Reusability, and High-Level Design. In *Proc. of the 1994 USENIX Symp. on Very High Level Languages,* October 1994.

[KV91] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of Summer USENIX Conference,* pages 235–256. USENIX, 1991.

[Pem92] S. Pemberton. The Ergonomics of Software Porting. Technical Report CS-R9266, Center for Mathematics and Computer Science of the Mathematical Centre Foundation, Amsterdam, December 1992.

[TC92] D. Tilbrook and R. Crook. Large scale porting through parameterization. In *Proceedings of the USENIX 1992 Summer Conference,* 1992.

[Vo90] Kiem-Phong Vo. IFS: A Tool to Build Application Systems. *IEEE Software,* 7(4):29–36, July 1990.

[WS90] Larry Wall and Randal Schwartz. *Perl.* O'Reilly & Associates, 1990.

Biography

Glenn Fowler is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of NMAKE, a configurable ANSI C preprocessor library, and the *coshell* network execution service. Glenn has been with Bell Labs since 1979 and has a B.S.E.E., M.S.E.E., and a Ph.D. in Electrical Engineering, all from Virginia Tech, Blacksburg Virginia.

David Korn received a B.S. in Mathematics in 1965 from Rensselaer Polytechnic Institute and a Ph.D. in Mathematics from the Courant Institute at New York University in 1969 where he worked as a research scientist in the field of transonic aerodynamics until joining Bell Laboratories in September 1976. He was a visiting Professor of computer science at New York University for the 1980-81 academic year and worked on the ULTRA-computer project (a project to design a massively parallel super-computer). Dave is currently a supervisor of research at Murray Hill, New Jersey. His primary assignment is to explore new directions in software development techniques that improve programming productivity. His best know effort in this area is the Korn shell, KSH, which is a Bourne compatible UNIX shell with many features added. The language is described in a book which he co-authored with Morris Bolsky. In 1987, he received a Bell Labs Fellow award.

John J. Snyder is a Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories, where he has done UNIX systems administration and now works mostly on software for end-user systems. He received a Ph.D. in Econometrics from the University of Colorado in 1979. At that time he worked with FORTRAN on a Cray-1 and UNIX on a DEC PDP 11/70 at the National Center for Atmospheric Research in Boulder. After consulting in Mexico City for a couple of years, he joined AT&T in 1983.

Kiem-Phong Vo is a Distinguished Member of Technical Staff in the Software Engineering Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include aspects of graph theory and discrete algorithms and their applications in reusable and portable software tools. Aside from obscure theoretical works, Phong has worked on a number of popular software tools including the curses and malloc libraries in UNIX System V, sfio, a safe/fast buffered I/O library and DAG, a program to draw directed graphs. Phong joined Bell Labs in 1981 after receiving a Ph.D. in Mathematics from the University of California at San Diego. He received a Bell Labs Fellow award in 1991.

Application Experience with an Implicitly Parallel Composition Language

R. Jagannathan and Chris Dodd
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, California 94025

{jaggan,dodd}@csl.sri.com

ABSTRACT

We describe our experiences with a very high-level parallel composition language (called GLU) that enables rapid construction of parallel applications using sequential building blocks (extracted from existing sequential applications) and their execution on diverse parallel computer systems. GLU is sufficiently rich to succinctly express different forms of parallelism — from function parallelism to data parallelism and from pipeline parallelism to tree parallelism. We show by example how a typical sequential application can be converted to a parallel one in GLU and executed on different parallel systems. We also show how GLU has been used to convert two widely used, sequentially written, inherently parallel workstation applications — the make utility and a raytracing system — to parallel equivalents that can then be run much faster on a network of workstations.

1 Introduction

The idea of composing new applications from existing ones is commonplace in various Unix[1] shells in which existing filters are often "plumbed" together in a pipeline fashion to form a new composite filter. For example,

```
ps -ax | grep -w "find" | sort | head -1
```

returns the process status description of the alphabetically first process with name `find`. This filter pipeline uses four other existing filters — `ps`, `grep`, `sort`, and `head` — and the plumbing is between the output of `ps` and the input of `grep`, output of `grep` and input of `sort`, and output of `sort` and input of `head`. The filter pipeline execution is basically data driven. However, the amount of data needed to start (and stop) execution varies with each filter — filter `ps` executes first and when it produces its first output filter `grep` also starts executing. Soon after filter `ps` produces its last output and then ceases execution, filter `grep` also stops. Only when filter `grep` produces all its output does filter `sort` perform its function. And as soon as filter `sort` produces its first output, filter `head` starts and completes its execution.

[1] All product names mentioned in this paper are the trademarks of their respective holders.

Latent in the filter pipeline is parallelism, in which different filters can be executing at the same time. In the above example, filters `ps` and `grep` can be simultaneously active, although filters `sort` and `head` can be active only after both `ps` and `grep` have ceased execution. Such latent parallelism can be exploited when different filters execute on different processors as would be the case when running on a symmetric multiprocessor.

The filter pipeline is an example of an *implicitly parallel* filter composition. Because a filter pipeline is linear and acyclic, it can express only one kind of parallelism — namely, *pipeline* parallelism. The Unix pipeline, which is character oriented, is typically implemented using file I/O, making flow of data through the pipeline much slower.

In this paper, we first review a parallel programming system based on the GLU (pronounced "glue") composition language for constructing parallel application using building blocks of existing sequential code. The language is similar to Unix filter pipelines in its use of dataflow-style plumbing to connect independent functions. We then describe our experiences with constructing parallel GLU applications, from existing sequentially written applications. We show how a prototypical sequentially written application with inherent parallelism can be converted into a GLU application, therein describing the various forms of parallelism that can be expressed using GLU. We then show how GLU can be used to rapidly convert two widely used sequentially written workstation applications — the **make** utility and a raytracing system — into parallel equivalents. We further demonstrate the benefit of doing so by considering application speedup on a network of workstations.

2 The GLU Parallel Programming System

We describe the GLU parallel programming system by first describing the GLU language and then describing how GLU programs are translated to parallel executables. We also briefly consider how GLU compares with related approaches.

2.1 GLU Composition Language

We have developed a composition language [5, 4]), which can be thought of as a generalization of filter pipelines, to enable rapid construction of parallel applications using existing, possibly sequential, functions[2]. The language is sufficiently rich to express different forms of parallelism, including function, data, pipeline, and tree parallelism.

An implicitly parallel composition in GLU consists of two parts: the first part declares the input and output interfaces of sequential functions to be used, and the second part declares the manner in which existing filters are "plumbed" together to form the parallel filter. GLU provides a set of predefined operators that enable arbitrary implicitly parallel composition of filters to be succinctly expressed.

The following illustrates a GLU composition for the above example:

```
string head( string, int );
string sort( string );
string grep( string, string );
string ps_ax();

head( sort( grep( ps_ax(), "find") ), 1 )
```

The filter functions used in the GLU composition are `head`, `sort`, `grep`, and `ps_ax`, whose input and output interfaces are defined in the first part of the program. The second part is a simple linear composition of the filters that corresponds to the Unix filter pipeline described above.

GLU compositions can express more than simply linear compositions. They can express general function compositions with latent structural parallelism. They can also express recursive function

[2] A sequential function in GLU is like a Unix filter in that its output depends only on its inputs.

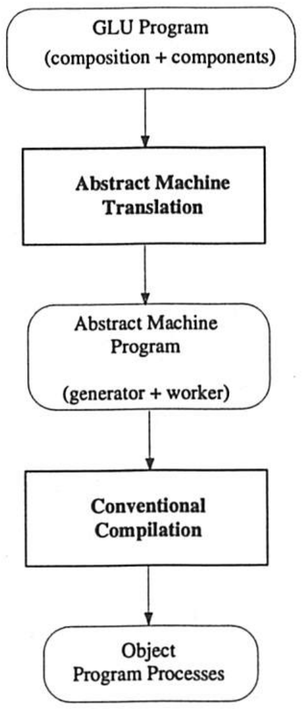


Figure 1: GLU Programming System Architecture

compositions with latent dynamic structural parallelism.

GLU is more than a functional language in which functions can be connected in different ways. GLU is also a multidimensional stream processing language. In the simplest case, a GLU expression denotes an infinite sequence of values in time. For example, the expression `ps_ax()` denotes the stream resulting from invoking `ps_ax` repeatedly at consecutive logical time steps. It is analogous to the Unix filter `yes` that repeatedly returns its argument over time. Even though all GLU streams are conceptually infinite, a special end-of-data value `eod` is used to denote the end of finite streams. GLU provides special operations that manipulate streams such as `fby` for combining two streams, `next` to drop the first value of a stream, `wvr` to select values from a stream, and `upon` to fill out a stream. These operations work not only on temporal streams but also on multidimensional streams. The stream processing capabilities of GLU can be used to express cyclic compositions as well as data-parallel compositions.

One of the high-level aspects of GLU is that GLU compositions express parallelism implicitly. This unburdens the programmer from having to explicitly manage parallelism on a per-application basis. The parallelism in GLU compositions is exploited using a novel model of execution — namely, demand-driven execution (also known as *eduction*) [1, 2]. The demand-driven model is general in that all forms of parallelism can be discovered and exploited. Unlike data-driven execution, demand-driven execution only causes useful filter functions to be invoked, thereby avoiding superfluous computations.

2.2 Translation of GLU Programs

The architecture of the GLU programming system is shown in Figure 1.

The GLU program is translated to an abstract machine, one that embodies the eduction model of computation. The abstract machine program representation of the GLU program consists of two parts: generator and worker. The generator consists of an application-indecedent abstract machine code interpreter, the abstract machine code for the GLU composition being translated, sequential functions to be locally invoked, and stubs for remotely invoking sequential functions. The worker consists of sequential functions that will be invoked by the generator with suitable wrappers. The interaction between generator and workers is assumed to be using a high-level remote procedure call facility.

The generator-worker representation of the GLU program can be compiled to a variety of parallel architectures — including workstation network, shared-memory multiprocessor, and massively parallel processors — provided the remote procedure call facility is available on the individual architectures. This compilation is conventional since the generator and the worker are C programs.

Executing a compiled GLU program requires execution of the generator and invocation of one or more workers. The generator continually generates executable functions (functions with available arguments) by interpreting the GLU composition using the abstract machine code interpreter. These functions are executed at the remote workers and the results are passed back to the generator. Note that it is possible to specify at compile-time or run-time that a function should be executed in the generator itself — functions so specified are executed in the generator while all other functions are executed at the worker. The principal source of parallelism is the simultaneous invocation of functions in multiple workers.

2.3 Related Approaches

GLU offers a high-level alternative to explicit, low-level approaches to parallel programming. Notable among the low-level approaches are PVM [8], Linda [3], and varieties of remote procedure call and message-passing packages [9]. The essential difference between GLU and these approaches is that GLU unburdens the programmer from the responsibility of discovering and managing parallelism. Thus, issues such as partitioning, mapping, scheduling, and load balancing are largely transparent to the programmer. Without this transparency, parallel programming becomes quite difficult as the programmer is forced to deal with issues at different levels of abstraction at the same time. The main criticism of high-level approaches such as GLU is that they are not efficient when compared to their low-level counterparts. Our experience with prototypical parallel applications suggests otherwise. Besides, we believe that the success of parallel programming will be initially determined more by how easy it is to build applications and less by how efficient they are.

3 Construction of a GLU Application: An Example

Consider the problem of multiplying two matrices, say A and B, and obtaining a product matrix C. The standard solution is to obtain each element (i, j) of the product matrix C by multiplying the i^{th} row of A with the j^{th} column of B and summing the inner product. A procedural program (in the language C) for multiplying matrices is as follows:

```
#include <stdio.h>

struct matrix {
        int nrows, ncols;
        float **v;
        };
typedef struct matrix *MATRIX;


MATRIX block( MATRIX M, int frow, int lrow, int fcol, int lcol )
{
  MATRIX m;
  int i, j;

  m = (MATRIX) malloc( sizeof( struct matrix ) );
  m->nrows = lrow - frow + 1;  m->ncols = lcol - fcol + 1;
  m->v = (float **) malloc( m->nrows * sizeof( float * ) );
  for( i=0; i<m->nrows; i++ )
    m->v[i] = (float *) malloc( m->ncols * sizeof( float ) );
  for( i=0; i<m->nrows; i++ )
    for( j=0; j<m->ncols; j++ )
      m->v[i][j] = M->v[frow+i][fcol+j];

  return( m );
}

MATRIX mult( MATRIX a, MATRIX b )
{
  MATRIX m;
  int i, j, k;
  float s;

  m = (MATRIX) malloc( sizeof( struct matrix ) );
  m->nrows = a->nrows;  m->ncols = b->ncols;
  m->v = (float **) malloc( m->nrows * sizeof( float * ) );
  for( i=0; i<m->nrows; i++ )
    m->v[i] = (float *) malloc( m->ncols * sizeof( float ) );

  for( i=0; i<m->nrows; i++ )
    for( j=0; j<m->ncols; j++ )
      {
        s = 0;
        for( k=0; k<a->ncols; k++ )
          s = s + a->v[i][k]*b->v[k][j];
        m->v[i][j] = s;
      }

  return( m );

}

MATRIX matmult( MATRIX A, MATRIX B )
{
 int i, j, nr, nc;
 MATRIX C, r;

 nr = A->nrows;
 nc = B->ncols;
 C = (MATRIX) malloc( sizeof( struct matrix ) );
 C->nrows = nr; C->ncols = nc;
 C->v = (float **) malloc( nr*sizeof( float * ) );
 for( i=0; i<nr; i++ )
   C->v[i] = (float *) malloc( nc*sizeof( float ) );

 for( i=0; i<nr; i++ )
   for( j=0; j<nc; j++ )
     {
        r = mult( block(A,i,i,0,nc-1), block(B,0,nr-1,j,j) );
        C->v[i][j] = r->v[0][0];
     }

 return( C );
}

main()
{
  MATRIX A, B, C;
```

```
    MATRIX inA(), inB();
    void out();

    A = inA();
    B = inB();
    C = matmult( A, B );
    out( C );
  }
```

The only user-defined datatype in the above program is **MATRIX**, which is a pointer to the structure `matrix` that consists of the number of rows, number of columns, and a two-dimensional array of `floats`.

In the above program, functions **inA** and **inB** are used to input matrices and **out** is used to output the product matrix; function **block** is used to extract a submatrix from a matrix; function **mult** is used to multiply compatible submatrices; and function **matmult** uses these functions to multiply two matrices.

We can develop a variant of function **matmult** as follows:

```
MATRIX gather( MATRIX tl, tr, bl, br )
{
  MATRIX m;
  int i, j, hnr, hnc;

  m = (MATRIX ) malloc( sizeof( struct matrix  ) );
  m->nrows = tl->nrows*2; m->ncols = tl->ncols*2;
  m->v = (float **) malloc(m->nrows*sizeof(float *));
  for(i=0; i<m->nrows; i++)
    m->v[i] = (float *) malloc(m->ncols * sizeof(float));

  hnr = m->nrows / 2;
  hnc = m->ncols / 2;

  for(i=0; i<m->nrows; i++)
    for(j=0; j<m->ncols; j++)
      {
        if( i < hnr )
          { if( j < hnc )
              m->v[i][j] = tl->v[i][j];
            else
              m->v[i][j] = tr->v[i][j-hnc];
          }
        else
          { if( j < hnc )
              m->v[i][j] = bl->v[i-hnr][j];
            else
              m->v[i][j] = br->v[i-hnr][j-hnc];
          }
      }

  return( m );
}

MATRIX lefthalf( MATRIX A )
{
  return( block( A, 0, A->nrows-1, 0, A->ncols/2 - 1 ) );
}
```

```
MATRIX righthalf( MATRIX A )
{
  return( block( A, 0, A->nrows-1, A->ncols/2, A->ncols-1 ) );
}

MATRIX tophalf( MATRIX A )
{
  return( block( A, 0, A->nrows/2 - 1, 0, A->ncols-1 ) );
}

MATRIX bottomhalf( MATRIX A )
{
  return( block( A, A->nrows/2, A->nrows-1, 0, A->ncols-1 ) );
}

MATRIX matmult( MATRIX A, MATRIX B )
{
 MATRIX C, TL, TR, BL, BR;

 TL = mult( lefthalf( A ), tophalf( B ) );
 TR = mult( righthalf( A ), tophalf( B ) );
 BL = mult( lefthalf( A ), bottomhalf( B ) );
 BR = mult( righthalf( A ), bottomhalf( B ) );

 C = gather( TL, TR, BL, BR );

 return( C );
}
```

In this program, multiplying a matrix consists of simply gathering the submatrices obtained by multiplying the appropriate half-matrices. The gathering function is performed using function `gather`, while selecting the appropriate half-matrices is performed using functions `lefthalf`, `righthalf`, `tophalf`, and `bottomhalf`, each of which is defined in terms of `block`.

The following routine constructs this program:

```
struct matrix {
        int nrows, ncols;
        float **m;
        }
typedef matrix *MATRIX;

int out( MATRIX );
MATRIX inA( );
MATRIX inB( );
MATRIX gather( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX lefthalf( MATRIX );
MATRIX righthalf( MATRIX );
MATRIX tophalf( MATRIX );
MATRIX bottomhalf( MATRIX );
MATRIX mult( MATRIX, MATRIX );

out( C ) fby eod where
  A = inA( );
  B = inB( );
  C = matmult( A, B );

  matmult( A, B ) = P where
     P = gather( TL, TR, BL, BR );
```

```
        TL = mult( lefthalf( A ), tophalf( B ) );
        TR = mult( righthalf( A ), tophalf( B ) );
        BL = mult( lefthalf( A ), bottomhalf( B ) );
        BR = mult( righthalf( A ), bottomhalf( B ) );
    end;
  end
```

The GLU program has two parts – the first part consists of user-defined datatypes and user-defined function-prototype declarations, and the second part consists of the composition of the program using built-in and user-defined functions. The syntax of the first part is identical to ANSI C, and hence needs no further explanation. The composition is simply an expression, in this case `out( C ) fby eod` where the enclosing *where clause* defines the variable `C`.

The expression `out(C) fby eod` defines a temporal sequence whose first element is the result of invoking `out(C)`, which has the side-effect of displaying matrix `C`, and whose subsequent elements are all `eod`, which is a special value denoting end of data. With most GLU interpreters, the operational effect of displaying an `eod` is termination of program execution.

The definition of `C` is given in the enclosing *where clause* as `matmult( A, B )`, where variables `A` and `B` themselves are defined using user-defined function `in`. Unlike `out` and `in` (which are C functions), `matmult` is a user-defined GLU function with two parameters. Function `matmult` is defined to be the variable `P` where `P` itself is the result of applying `gather` to `TL`, `TR`, `BL`, and `BR`. Variable `TL` is the result of multiplying (using `mult`) the left half of matrix `A` (obtained using `lefthalf( A )`) and the top half of matrix `B` (obtained using `tophalf( B )`). Variables `TR`, `BL`, and `BR` are similarly defined.

It is worth noting how this composition is evaluated, given that it is declarative and not procedural. The first thing that happens is that expression `out(C) fby eod` is evaluated. It causes `out(C)` to be evaluated first and `eod` to be evaluated next. Since `out` is a procedural user-defined function, it is invoked only when variable `C` is defined and no sooner. To evaluate `C`, the right-hand side of its definition, `matmult( A, B )`, is evaluated. Since `matmult` is a composition function and not a procedural function, it is evaluated even before its arguments `A` and `B` are available. In fact, all composition functions are evaluated using "call-by-need" semantics as opposed to "call-by-value" semantics used for C functions.

The main difference between a GLU program and an equivalent procedural program is that the GLU program is inherently parallel unless otherwise constrained, whereas the procedural program is inherently sequential unless parallelism is explicitly identified. In the example above, with the GLU program, the variables `TL`, `TR`, `BL`, and `BR` can be computed simultaneously, resulting in fourfold parallelism, whereas these variables have to be evaluated sequentially in the procedural program.

Since GLU programs are declarative, they possess the important property of *referential transparency*, which means that two occurrences of an expression refer to the same value. Thus, the order of equations in GLU programs is simply a matter of style and does not affect their meanings. One can substitute the right-hand side of an equation for each occurrence of the left-hand side and vice versa, just as in mathematics.

GLU programs separate composition from computation. This is important in various aspects of the program development process, such as debugging and modification, and also encourages reuse of existing computational code across applications.

The GLU program described above simply multiplies two matrices and then terminates. What if we want to have the program multiply two streams of matrices pairwise? It turns out that doing so in GLU is not that much more work than multiplying just one pair. The GLU composition for this procedure is

```
struct matrix {
        int nrows, ncols;
        float **m;
        }
typedef matrix *MATRIX;

int out( MATRIX );
MATRIX inA( int );
MATRIX inB( int );
MATRIX gather( MATRIX, MATRIX, MATRIX, MATRIX );
MATRIX lefthalf( MATRIX );
MATRIX righthalf( MATRIX );
MATRIX tophalf( MATRIX );
MATRIX bottomhalf( MATRIX );
MATRIX mult( MATRIX, MATRIX );

out( C ) where
  A = inA( #.time );
  B = inB( #.time );
  C = matmult( A, B );

  matmult( A, B ) = P where
     P = gather( TL, TR, BL, BR );
     TL = mult( lefthalf( A ), tophalf( B ) );
     TR = mult( righthalf( A ), tophalf( B ) );
     BL = mult( lefthalf( A ), bottomhalf( B ) );
     BR = mult( righthalf( A ), bottomhalf( B ) );
  end;
end
```

This program differs from the previous one in only two respects: functions `inA` and `inB` now have an argument, `#.time`, and the program expression is simply `out(C)` instead of `out(C) fby eod`.

What this program reveals is the underlying context in which GLU programs are evaluated — namely, `time`. That is, each expression (including variables) in the above GLU program actually denotes a temporal stream of values. The value of an expression depends not only on the constituent subexpressions and their binding function but also on the implicit time context. Thus, one can think of the equation `C = matmult( A, B )` as meaning

$<C_0,\ldots,C_t,\ldots> = <\texttt{matmult(A,B)}_0,\ldots,\texttt{matmult(A,B)}_t,\ldots>$
where $\texttt{matmult(A,B)}_t = \texttt{matmult(}\ A_t,\ B_t\ \texttt{)}$.

Consider the evaluation of $\texttt{out(C)}_t$: it causes C_t to be evaluated, which causes $\texttt{matmult(A,B)}_t$ to be evaluated, which, in turn, causes $\texttt{gather(}TL_t,TR_t,BL_t,BR_t\texttt{)}$ to be evaluated. Evaluations of TL_t,TR_t,BL_t, and BR_t eventually cause both A_t and B_t to be evaluated once. Evaluation of A_t invokes `inA` with `#.time` being the current time context — namely, t. (And similarly for B_t.) Once A_t and B_t are available, TL_t,TR_t,BL_t, and BR_t can be generated, resulting in C_t to be generated and function `out` to display the matrix.

Since evaluation of C_t is independent of C_{t+1}, it is entirely possible for both these computations and any others to proceed simultaneously without interference.

Suppose we wanted to multiply only the first 10 matrix pairs and then terminate – this requires changing only the program expression from `out(C)` to `if #.time < 10 then out(C) else eod fi`.

One of the limitations of `matmult` is that it results in only fourfold parallelism since function `mult` is sequential. It is possible to have `matmult` express more parallelism by making `matmult` recursive:

```
  matmult( A, B ) =
    if smallenough(A,B)
```

```
          then mult(A,B)
          else P fi where
           P = gather( TL, TR, BL, BR );
           TL = matmult( lefthalf( A ), tophalf( B ) );
           TR = matmult( righthalf( A ), tophalf( B ) );
           BL = matmult( lefthalf( A ), bottomhalf( B ) );
           BR = matmult( righthalf( A ), bottomhalf( B ) );
        end;
```

In this program, we have introduced a function **`smallenough`** that helps control the granularity of parallelism by deciding when multiplying two matrices cannot be further subdivided. Composition function `matmult` itself is recursive – instead of having `TL`, `TR`, `BL`, and `BR` defined in terms of `mult`, these variables are defined using `matmult`. The parallelism expressed by `matmult` is no longer fourfold — it is actually $O(4^L)$, where L is the number of recursive steps.

A drawback of the recursive `matmult` is the cost associated with repeated division of matrices using `lefthalf`, `righthalf`, `tophalf`, and `bottomhalf`. This not only takes time but also causes unnecessary creation and destruction of intermediate matrices. We devise a nonrecursive version of `matmult` that expresses as much parallelism, without the cost of recursion. To do so, we use the GLU notion of user-defined multidimensionality:

```
matmult( A, B ) = p asa.t ( (nrows(A) == nrows(p)) && (ncols(B) == ncols(p)) )
  where
    dimension i,j,t;
    a = rowstrip( A, #.i );
    b = colstrip( B, #.j );
    p = mult( a, b ) fby.t
        ( ( gather( p, next.j p, next.i p, next.i next.j p )
        @.j (2*#.j) ) @.i (2*#.i) );
  end;
```

Composition function `matmult` is defined using a multidimensional computation in dimensions `i`, `j`, and `t`, which are declared using the `dimension` declaration within the *where clause* associated with `matmult`. This multidimensional computation occurs at each `time`-context in the enclosing `time` dimension. It is defined to be the value of variable `p` at context `i=0,j=0,t=T` where `T` is the `t`-context at which the number of rows of the product matrix `p` is the same as the number of rows of matrix `A` and the number of columns is the same as the number of columns of matrix `B`.

Consider the definition of variable `p`, which says that at any context where `t`-context is 0, it is the value of `mult(a,b)` at the same context. Variable `a`, which varies only in dimension `i`, is defined as function `rowstrip(A, #.i)`, where `#.i` refers to the `i`-context from the context of evaluation. (`rowstrip` basically extracts the $\mathtt{i}^{th}$ strip of rows from matrix `a`.) Variable `b`, which varies only in dimension `j`, is defined as function `colstrip(B, #.j)` where `#.j` refers to the `j`-context from the context of evaluation. (It extracts the $\mathtt{j}^{th}$ strip of columns from matrix `b`.)

Variable `p` at context `(i=I,j=J,t=T)` where `T>0` is the result of applying function `gather` to values of variable `p` at contexts `(i=2*I,j=2*J,t=T-1)`, `(i=2*I,j=2*J+1,t=T-1)`, `(i=2*I+1,j=2*J,t=T-1)`, and `(i=2*I+1,j=2*J+1,t=T-1)`. This can be derived as follows:

$$
\begin{aligned}
&\mathtt{P}_{(\mathtt{i=I,j=J,t=T})} = \\
&\qquad \mathtt{(gather(p,next.j\ p,next.i\ p,next.i\ next.j)@.j(2*\#.j))@.i(2*\#.i)}_{(\mathtt{i=I,j=J,t=T-1})} \\
&\quad = \mathtt{gather(p,next.j\ p,next.i\ p,next.i\ next.j)}_{(\mathtt{i=2*I,j=2*J,t=T-1})} \\
&\quad = \mathtt{gather(p}_{(\mathtt{i=2*I,j=2*J,t=T-1})}, \mathtt{P}_{(\mathtt{i=2*I,j=2*J+1,t=T-1})}, \\
&\qquad\quad \mathtt{P}_{(\mathtt{i=2*I+1,j=2*J,t=T-1})}, \mathtt{P}_{(\mathtt{i=2*I+1,j=2*J+1,t=T-1})}\mathtt{)}
\end{aligned}
$$

The best way to visualize the evaluation of `matmult` is to think of each of its variables as representing a series of planes `(i,j)` ordered by time `t`. Variable `a` then is a series of identical planes (since the definition of a does not depend on time t) where each plane consists of identical columns of row strips where each row strip is extracted from matrix a. Similarly, variable b is a series of identical planes where each plane consists of identical rows of column strips where each column strip is extracted from matrix b. (The row strip size and column strip size are predefined.) It is worth noting that any reasonable implementation of GLU would store values of a for each i-context and values of b for each j-context, as they are constant in the remaining two dimensions.

Variable p consists of a series of planes, where the initial plane consists of matrix products such that the $(i,j)^{th}$ matrix is obtained by multiplying the corresponding (actually i^{th}) row strip of a and corresponding (actually j^{th}) column strip of b. Each successive plane consists of matrices that are four times as large where the $(i,j)^{th}$ matrix is obtained by coalescing the four matrices using function gather in the previous plane at positions $(2i, 2j)$, $(2i, 2j+1)$, $(2i+1, 2j)$, and $(2i+1, 2j+1)$.

Assuming that the size of each element matrix in the initial plane is s and the size of the product matrix is S, the $(0,0)^{th}$ element of the $\log_2(S/s)$ plane is the desired product matrix. One way of determining without keeping track of size is to check whether the $(0,0)^{th}$ element matrix of a plane has the same number of rows as matrix a and the same number of columns as matrix b, as in p asa.t ((nrows(A) == nrows(p)) && (ncols(B) == ncols(p))).

Similar to the recursive version of matmult, the parallelism in simultaneous execution of mult is 4^L where L is the number of planes. (L is $\log_2(S/s)$.)

It turns out that this kind of computational structure is common to algorithms that solve quite different problems. Therefore, it would be desirable to specify the structure in such a way that it can be not only general but also succinct. This is the idea behind what we call dimensionally abstract composition functions — functions that abstract not only data but also dimensions. We illustrate one such function, planar_tree, in matmult:

```
planar_tree.x,y( f, a, size ) = b asa.t s >= size
  where
    dimension t;
    b = a fby.t
        ( ( f( b, next.y b, next.x b, next.y next.x b )
        @.x (2 * #.x)) @.y (2 * #.y));
    s = 1 fby.t 4*s;
  end;
matmult( A, B ) = planar_tree.i,j( gather, mult(a, b), n )
  where
    dimension i,j;
    a = rowstrip( A, #.i );
    b = colstrip( B, #.j );
    n = ( nrows(A) / ROWSTRIPSIZE ) * ( nrows(B) / COLSTRIPSIZE );
  end;
```

In matmult, planar_tree is invoked with dimensions x and y corresponding to i and j, f corresponding to data function gather, a corresponding to mult(a,b), and m corresponding to n.

4 Application Experience

In this section, we describe our experience in converting two existing sequential applications to GLU and their resulting performance improvement when executed on a network of workstations. It is worth noting that other sequential applications have similarly been converted to GLU including a 400,000 line message handling system [6] and a large structural engineering code [7, 2].

4.1 Parallel Make

Make, a widely used Unix utility, is a natural candidate for parallelization. In fact, there are several commercial and public domain offerings of parallel **make**. The idea in **make** is to use a set of acyclic dependencies between program objects (usually files) to determine whether a target object is current; if the target object is not current, to use the associated actions to build a current target object.

This obviously is a recursive definition as each such object, in turn, is a target with its own set of dependencies. Associated with each dependency is a set of actions that builds a target if it is not current.

We describe the parallelization of **make** (specifically **GnuMake**) by converting to GLU to illustrate the rapidity and effectiveness with which we were able to do so. (Interestingly, the **make** algorithm and the GLU model of computation have much in common — both are demand-driven and both operate on data-dependency networks.)

We started with **GnuMake**, which is a sequentially written application from Free Software Foundation. It consists of over 15,000 lines of C code in 35 files. The principal source of parallelism is in being able to build multiple independent targets simultaneously. This parallelism is latent in the makefile itself. In fact, **GnuMake** provides the capability to exploit this parallelism on a shared-memory multiprocessor by starting multiple processes to build independent targets.

We are interested in creating a parallel make that exhibits the same behavior as **GnuMake**. And we want to do so without writing large amounts of new code. Our approach is simple to describe. We modified **GnuMake** to generate dependency information of the targets that would have been built without actually building them. This information is piped to a GLU program that uses it to construct the dependency tree. Starting with the root of the tree, it applies the associated build actions after each of its immediate ancestors have been successfully built and this proceeds recursively. The GLU program composition is shown below.

```
typedef int CODE;
typedef int STATUS;
typedef struct cmds {
    int         count;
    string      line[count];
    } *CMDS;
typedef struct object {
    string      name;
    int         no_dependents;
    CMDS        commands;
    void        *treeptr;
    } *OBJECT;
local int       input_from_make( int );
local OBJECT    getobject( int );
local int       getancestor( int, int );
      STATUS    apply( OBJECT, CODE );

M( input_from_make( 0 ) ) fby eod
where
  dimension target;
  M( f ) = if iseod f then -1 else make fi;
  make = if errcode then errcode else apply(object, errcode) fi;
  object = getobject( #.target );
  errcode = done where
    dimension sibling;
    done = if ( iseod ancestors ) then 0
           else (make @.target ancestors) | next.sibling done fi;
    ancestors = if a<0 then eod else a fi where
            a = getancestor(#.target, #.sibling);
        end;
    end;
end
```

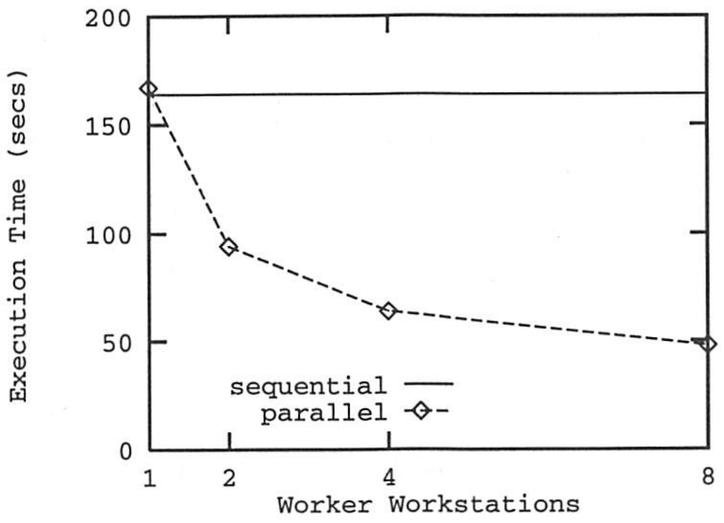


Figure 2: Performance of Parallel GLU Make

It uses four C functions `input_from_make`, `getobject`, `getancestor` and `apply`.. Function `input_from_make` constructs a dependency tree using information provided by a benign invocation of **GnuMake**. The dependency tree is kept in a global data structure (actually, a hash table) accessible to `local` functions i.e., ones that execute in the generator itself. Function `getobject` returns an `OBJECT` data structure pointed to by its argument from the hashtable. Function `getancestor` returns the index of the appropriate ancestor of the current target (given by `#.target`). Function `apply` builds the current target using actions given in its first argument. It is the only function executed in the workers; all other functions being `local` are executed in the generator itself.

The program corresponds to GLU function `M(input_from_make(0))` which has the effect of building the dependency tree in a hashtable and initiating the make process starting with the root target (at `#.target=0`). For `apply(object,errcode)` to be invoked at context `#.target=0`, `getobject` and `errcode` at the same context must be evaluated. Function `getobject` will return immediately but evaluation of `errcode` requires further evaluation. In particular, it requires all the immediate ancestors of target at `#.target=0` to be evaluated in the same way. And this proceeds recursively to the leaves of the dependency tree. At a leaf target, since there are no ancestors, its `errcode` can be immediately computed causing its `apply` to be invoked. The results of such `apply` invocations will flow back to other suspended `apply` invocations, and eventually to the one associated with the root target (`#.target=0`).

The conversion only required adding 15 lines of C code and changing 2 lines of C code in GnuMake. The GLU make program itself is less than 40 lines of composition and 250 lines of new C code for the sequential functions declared in the composition.

Figure 2 shows the performance of parallel GLU make applied to a typical makefile (for **GnuEmacs** generation) on a ethernet-based network of lightly loaded, similarly configured Sun SS2 workstations with a common file system. The speedup using parallel make is best seen for a small number of workstations. It drops off as the number of workstations increase. This is partly because the granularity of parallelism (ratio of remote function time to communication overhead) is small and variable (as builds of different targets take different times). The other reason is that the root of the dependency tree has a monolithic build that takes sizable time (25 seconds in the experiment). This adversely affects the extent to which performance can scale with additional workstations.

4.2 Parallel Raytracing

Raytracing is a widely used graphics application that is inherently highly parallel. The basic idea in raytracing is to render a two-dimensional view of an observer of a 3D scene consisting of objects with different reflective and refractive properties and multiple sources of light. The two-dimensional view is a pixel plane where the color (RGB value) of each pixel is determined by the raytracing algorithm. The standard algorithm is such that the RGB value of each pixel can be computed independent of the others.

The sequential implementation of raytracing that we started with consisted of approximately 3000 lines of C code over 25 files. The `main` function of the implementation processes the various command line options, initializes global data structures, reads the scene data, and invokes the function `Screen` to perform raytracing. This function calls one of three slightly different functions, each of which essentially consists of a doubly nested loop that computes the RGB value of each pixel in the two-dimensional image using function `Trace`.

The principal source of parallelism in this application is in the unraveling of the doubly nested loop such that each iteration can essentially proceed independently. However, invoking each iteration independently has associated overhead that can obviate the benefits of doing so. Therefore, it is necessary to granulate the parallelism by making each iteration serially compute RGB values of several pixels while reducing the number of iterations. Although this conceptually reduces parallelism, it reduces the overhead per pixel, which improves the effectiveness with which parallelism is exploited.

In converting this application to GLU, we expose the parallel structure of raytracing in the composition part while subsuming the per-pixel processing in sequential functions. The GLU program composition for the raytracing application is

```
struct INPUT
{
  int x_dimension;
  int y_dimension;
  int grainsize;
  char name[64];
};
typedef struct INPUT INPUT;
struct STRIP
{
  int cols, rows;
  unsigned char line[cols][rows][3];
};
typedef struct STRIP STRIP;
local int display( string, int, int, int, int, int, int, STRIP *);
      STRIP *raytrace( int, int, int, int, int, int, int, int, int );
local INPUT *input( int );
local int xdim_in( INPUT *);
local int ydim_in( INPUT *);
local int grainsize_in( INPUT *);
local string appl_name_in( INPUT * );
local int reset( int );

pics where
  pics = pic;
  pic = image( xdim, ydim, ntiles );
  image( xd, yd, n ) = reset( p )
```

```
      where
          p = planar_tree.x,y(
                  combine,
                  display( appl_name, x, y, xdim, ydim, ifactor, ifactor, tile )
                    where
                      tile = raytrace( time, x, y, xdim, ydim,
                                       ifactor, ifactor, xdim, ydim );
                      ifactor = sqrt( n );
                    end,
                   n )
                where
                  dimension x, y;
                  combine( a, b, c, d ) = a + b + c + d;
                end;
      end;
    ntiles = xdim * ydim / grainsize;
    xdim = xdim_in( inp );
    ydim = ydim_in( inp );
    grainsize = grainsize_in( inp );
    appl_name = appl_name_in( inp );
    inp = input( #.time ) fby pic,inp;
  end
```

We introduce two new type definitions: **INPUT** and **STRIP**. These are used by sequential functions that provide the interface between composition and existing sequential code. We define a function **input** that reads application parameters from **stdin**. Thus, the command-line interface to the application has changed. Four simple functions — **xdim_in**, **ydim_in**, **grainsize_in**, and **appl_name_in** — are defined, each of which extracts the appropriate field from the structure returned by **input**. We define function **display**, which when invoked with several arguments including an image, superimposes the argument image on an existing X window. Associated with **display** is function **reset** that creates a new X window to display images. Finally, we define function **raytrace**, which accepts nine integer arguments that help define the set of pixels of the image that it is responsible for computing using existing functions such as **Trace**. Also, **raytrace** is the only function that is executed in the workers; all other functions being **local** are executed in the generator itself.

The GLU composition defines a stream of images (in **pics**) where each image is denoted by the variable **pic**. This variable is defined as the GLU function **image** applied to **xdim**, **ydim**, and **ntiles** where these arguments are provided as, or derived from, **inp** (returned by **input**). GLU function **image** returns the vertex of a two-dimensional pyramid computation (specified by **p** shown in Figure 3).

Each invocation of **display(raytrace(...))** at the base of the pyramid computes and displays a set of pixels independent of any other invocation. The number of such invocations, and hence the degree of parallelism, is given by **ntiles**. The interior of the pyramid simply computes the sum of the values returned by the base invocations, although the result is not of any use to the application user. When the vertex of the pyramid has been computed, function **reset** is invoked, which creates a new X window to display images.

The new sequential code that we have generated corresponds to functions **input**, **xdim_in**, **ydim_in**, **grainsize_in**, **appl_name_in**, and **reset**, which together amount to less than 40 lines of C code and function **raytrace**, which is mostly a rearrangement of existing code (from functions **main** and **Screen**) with around 10 to 20 lines of new C code. The parallelization of the sequentially written raytracing application into GLU required around 2 to 3% new C code and approximately 60 lines of the GLU composition.

Figure 4 shows the performance of parallel raytracing constructing a series of 400x400 pixel image on a network of Sun SS2 workstations. It scales much better than parallel make and exhibits

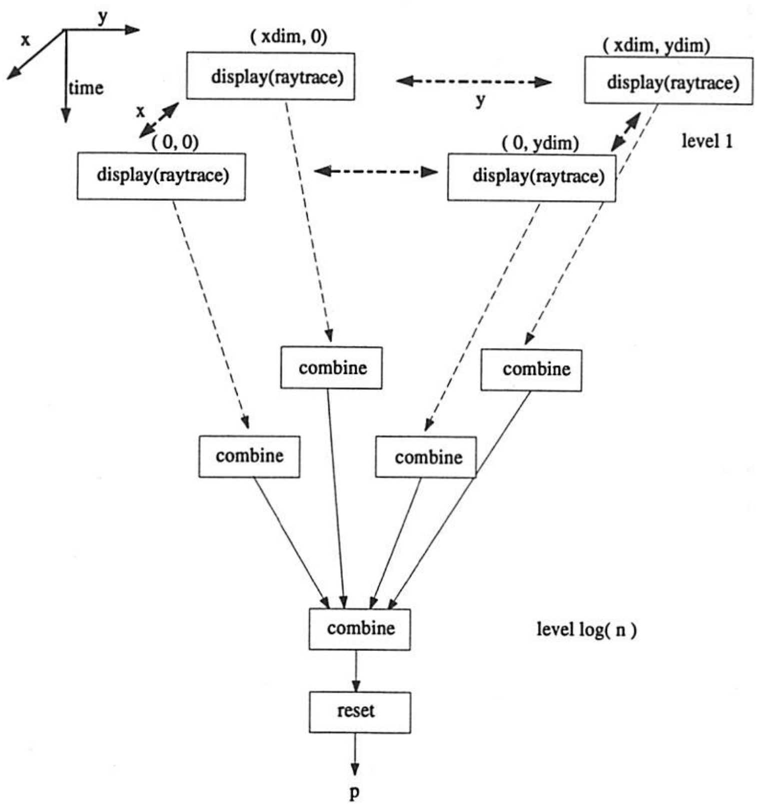


Figure 3: Two-Dimensional Raytracing Computational Pyramid

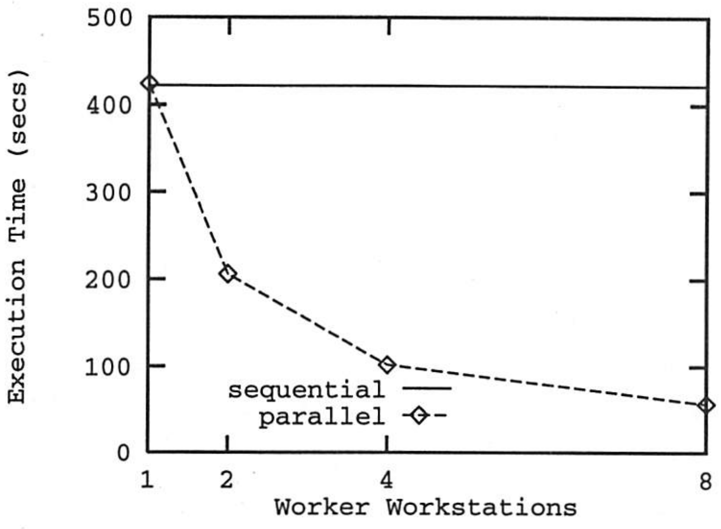


Figure 4: Performance of Parallel Raytracing

near-linear speedup for the number of workstations we considered. This application performs better than parallel make because the granularity of parallelism is much coarser and quite uniform. In addition, parallel make of the makefile we considered has a sizable sequential part as we noted earlier.

5 Conclusions

Our experience with a very high-level composition language called GLU demonstrates its rôle in constructing parallel applications using existing sequentially written codes. The expressiveness of the language is illustrated by the different ways in which a prototypical sequentially written, inherently parallel application (matrix multiplication) can be parallelized.

Two widely used, sequentially written applications — namely, **make** and raytracing — have been converted to parallel GLU equivalents with only a nominal amount of modifications and additions. Application speed has impressively improved by running parallel versions of the two applications on a network of workstations.

A very high-level language like GLU offers parallel programmers the ability to solve problems without getting mired in the often intricate details of how to manage parallelism; as is often the case when using low-level parallel languages. In addition, it enables existing sequential applications to be rapidly converted to parellel GLU applications by allowing most of the existing sequential code to be reused.

Acknowledgements

This work was supported in part by the National Science Foundation (Grant Number CCR-9203732) and in part by SRI's IR&D investment program.

References

[1] E.A. Ashcroft. Dataflow and Eduction: Data-driven and demand-driven distributed computation. In *Current Trends in Concurrency*. Springer Verlag, 1986. Lecture Notes in Computer Science (224).

[2] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Declarative Programming*. Oxford University Press, 1994.

[3] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, April 1989.

[4] R. Jagannathan and C. Dodd. GLU programmer's guide (version 0.9). Technical Report SRI-CSL-94-06, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, July 1994.

[5] R. Jagannathan and A.A. Faustini. GLU: A hybrid language for parallel applications programming. Technical Report SRI-CSL-92-13, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1992.

[6] Y. Koui, S. Seno, T. Yamauchi, M. Ishizaka, and K Kotaka. Implementation and evaluation of MHS parallel processing. *IEICE Transactions on Communications*, E77B(11), November 1994. Tokyo, Japan.

[7] E. Moreno. GLU BRIDGE—A structural engineering application in the GLU programming language. Master's thesis, Arizona State University, Tempe, Arizona 85287, U.S.A., 1993.

[8] V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[9] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a mechanism for intergrated communication and computation. In *Proceedings 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM Press, 1992.

Author Information

R. Jagannathan is a Senior Computer Scientist at SRI International in the Computer Science Laboratory. His interests include declarative languages, parallel and distributed computing, and adaptive systems. He received a B.Sc. from the University of Calgary in 1979 and a M.Math. and a Ph.D. from the University of Waterloo in 1981 and 1988 respectively.

Chris Dodd is a Senior Software Engineer at SRI International in the Computer Science Laboratory. His interests include compilation techniques, parallel and distributed operating systems, and languages. He received a B.S. from the California Institute of Technology in 1988 and a M.S. from Stanford University in 1990.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- fostering innovation and communicating both research and technological developments,
- providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its annual technical conference, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

* Subscription to *;login:*, a bi-monthly newsletter;
* Subscription to *Computing Systems*, a refereed technical quarterly;
* Discounts on various UNIX and technical publications available for purchase;
* Discounts on registration fees to the annual technical conference and tutorial programs and to the periodic single-topic symposia;
* The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
* The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO
ASANTÉ Technologies, Inc.
Frame Technology Corporation
Matsushita Electrical Industrial Co., Ltd.
OTA Limited Partnership
Quality Micro Systems, Inc.
SunSoft Network Products, Sun Microsystems, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

SAGE Supporting Member:

Enterprise System Management Corporation

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-65-0